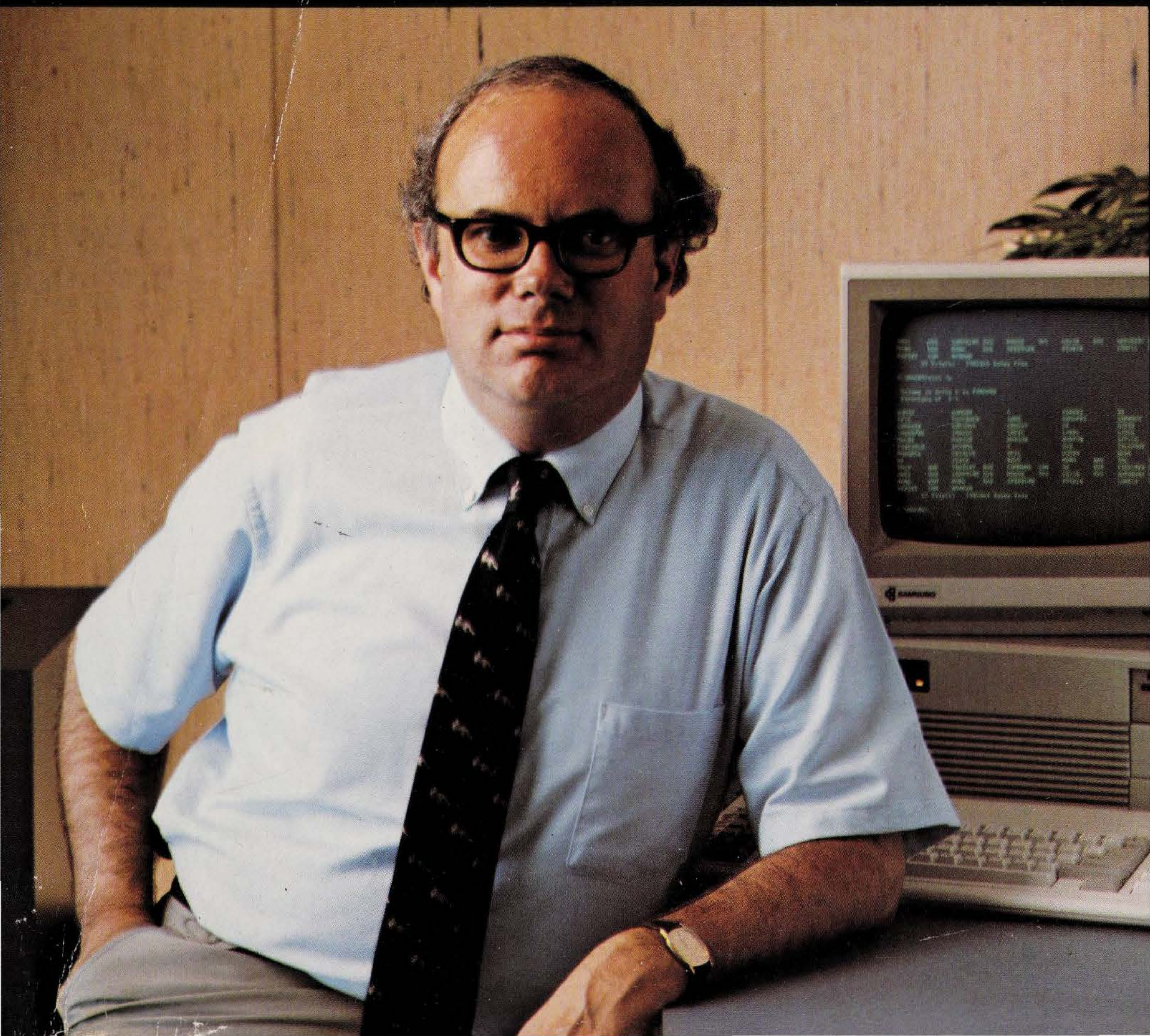




THE PC LIBRARY

LEVENTHAL'S 80386 PROGRAMMING GUIDE



LANCE LEVENTHAL'S

80386
PROGRAMMING
GUIDE

LANCE LEVENTHAL'S

80386 PROGRAMMING GUIDE

Lance
Leventhal



BANTAM BOOKS

TORONTO • NEW YORK • LONDON • SYDNEY • AUCKLAND

LANCE LEVENTHAL'S 80386 PROGRAMMING GUIDE
A Bantam Book / January 1988

LANCE LEVENTHAL'S 80386 PROGRAMMING GUIDE
A Bantam Book / January 1988
2nd printing . . . February 1988

All rights reserved.
Copyright © 1987 by Lance Leventhal.
Produced by Micro Text Productions.
No part of this book may be reproduced or transmitted
in any form or by any means, electronic or mechanical,
including photocopying, recording, or by any information
storage and retrieval system, without permission in
writing from the publisher.
For information address: Bantam Books.

ISBN 0-553-34529-X

Published simultaneously in the United States and Canada

Bantam Books are published by Bantam Books, a division of Bantam Doubleday Dell Publishing Group, Inc. Its trademark, consisting of the words "Bantam Books" and the portrayal of a rooster, is Registered in U.S. Patent and Trademark Office and in other countries. Marca Registrada. Bantam Books, 666 Fifth Avenue, New York, New York 10103.

PRINTED IN THE UNITED STATES OF AMERICA

B 11 10 9 8 7 6 5 4 3 2

Contents

Preface	xi
Chapter 1	
Introducing the 80386	1
Key Features	2
Virtual 8086 Mode	6
Pipelined Architecture	6
Typical Applications	9
Personal Computers	11
CAD/CAM/CAE Systems	14
Robotics	17
Artificial Intelligence	18
Signal Processing	19
Key Concepts	20
Virtual Memory	21
Multitasking	25
Multiuser Systems	27
Protection	28
Support for High-Level Languages	29
Comparisons with Previous Processors	29
What's Next in Microprocessors?	30
Summary	31
Chapter 2	
80386 Architecture and Instruction Set	33
Notation	34
Registers	34

Data Types	42
Addressing Modes	45
Instruction Set	48
Frequently Used Data Transfer Instructions	54
Frequently Used Arithmetic and Logical Instructions	58
Frequently Used Program Control Instructions	60
General Data Transfer Instructions	61
General Data Manipulation Instructions	63
General Program Control Instructions	71
Other Instructions	71
Address and Operand Size	75
Instruction Speedups	79
Instructions and Flags	79
8086 and 80286 Compatibility	80
Assembler Directives	81
Summary	84

Chapter 3

Assembly Language Programming 87

80386 Highlights	88
Simple Programs	89
Bit Manipulation	91
Shift Operations	92
Making Decisions	96
Looping	101
Array Manipulation	101
Table Lookup	103
Character Manipulation	104
Code Conversion	107
Multiple-Precision Arithmetic	108
Data Structure Manipulation	109
Parameter Passing Techniques	115
Making Programs Run Faster	117
Common Programming Errors	118
Summary	119

Chapter 4

Input/Output

121

Alternative I/O Methods	121
I/O Addressing	122
I/O Instructions	124
Programmable I/O Chips	125
8250 ACE	127
8255 PPI	130
8253 and 8254 PITs	132
I/O Examples	133
Interrupts	137
Sample Interrupt Service Routines	140
Interrupt Controller	143
Direct Memory Access	149
Summary	151

Chapter 5

80386 Memory Management

153

80386 Highlights	153
Memory Management	154
Operating Modes	155
Segmentation	157
8086 Segmentation Methods	157
Protected Mode Segmentation	161
Paging	167
Page Translation	167
Page Tables	171
Page Cache	173
Memory Protection	174
Domain Restrictions	178
Restricting Control Transfers	178
Conforming Code Segments	183
Creating Descriptors	183
Privileged Instructions	184

Initialization of Memory Management Systems	184
Summary	185

Chapter 6

80386 Task Management 187

What Is Tasking?	188
80386 Tasking Features	190
Task State Segments	192
Task State Segment Descriptors	195
Task Register	196
Task Gate Descriptors	197
Task Switching	199
Task Linking	201
Task Address Spaces	202
I/O Privilege Levels	203
I/O Permission Maps	203
Initialization of Tasking Systems	205
Summary	208

Chapter 7

80386 Exceptions and Debugging Features 211

New 80386 Features	212
Sources of Exceptions	212
Interrupt Descriptor Table	214
Error Codes	217
Exception Conditions	219
Invalid Operation Code Exception	220
Double Faults	220
Invalid Task State Segment Faults	223
Segment Not Present Exceptions	223
Stack Exceptions	224
General Protection Exceptions	224
Page Faults	225
Debugging Features	226
Debug Registers	227

Debug Exceptions	229
Summary	229

Chapter 8

80386 Hardware Features 231

New 80386 Features	231
80386 External Signals	232
Memory and I/O Transfer Control Signals	232
Startup Signals	239
Coprocesor Signals	241
Interrupt Control Signals	242
DMA Signals	242
80386 Bus Operation	242
Pipelined Bus Cycles	247
Interrupt Acknowledge Cycles	247
Bus Performance Considerations	249
Coprocessors	250
80287 Numeric Coprocessor Interface	250
80387 Numeric Coprocessor Interface	253
Local Coprocessor Bus Cycles	253
80287/80387 Recognition	254
Coprocesor Exceptions	255
Memory Interfacing	255
Cache Memory	260
Cache Controller	260
Block Size	261
Cache Organization	261
Direct Mapped Cache	264
Set Associative Caches	265
Cache Updating	265
Non-Cacheable Memory	267
Cache Performance	268
Summary	268

Appendix A 80386 Instruction Set	271
Appendix B Complete 80386 Flag Cross-Reference	295
Appendix C Status Flag Summary	297
Appendix D Summary of 80386 Descriptors	299
Appendix E Differences between 80286 and 80386 Processors	305
Appendix F Differences between 8086 and 80386 Processors	309
Appendix G Overview of the 80287 and 80387 Numerical Data Processors	315
Appendix H Instruction Set of the 80287 Numerical Data Processor	319
Appendix I Instruction Set of the 80387 Numerical Data Processor	325
Acknowledgements	331
Index	333

Preface

The 80386 microprocessor is the first 32-bit member of the popular Intel 8086 family. This family's applications include communications equipment, instrumentation, graphics, CAD/CAM, test equipment, industrial control, process control, business equipment, and military systems. Family members are also the CPUs in the IBM PC and PS/2 series, PC compatibles, and PC clones. Its early introduction, many associated devices, and extensive hardware and software support have helped the 8086 family dominate the microprocessor market.

This book is a general introduction to the 80386 for programmers, engineers, technicians, systems analysts, teachers, students, and personal computer users. It can also serve as a primer for computer and data processing professionals and instructors and as a supplemental text for courses in computer organization, microprocessors, personal computers, and computer applications. It emphasizes the 80386's key features and the differences between it and the earlier 8088, 8086, and 80286 chips. Examples illustrate the use of these features in typical applications and point out problems and pitfalls.

The book assumes that readers are familiar with the 8086 family and with a programming language such as BASIC, C, FORTRAN, or Pascal. Readers should also have some background in computer architecture and assembly language programming, derived either from course work or from practical experience.

The book is organized as follows:

Chapter 1 is a general overview of the 80386. It emphasizes new features and concepts, including support for virtual memory, multitasking, multiuser systems, operating systems, and high-level languages. It also compares the 80386 with previous processors, describes typical applications, and lists new features that the next generation of processors may have.

Chapter 2 presents the 80386's instruction set from the user's point of view. It emphasizes the major features of the processor's architecture and addressing modes. It then focuses on frequently used instructions before dealing with the entire instruction set. It also discusses the 80386's improved performance, the effects of instructions on flags, differences between the 80386 and earlier processors, and assembler directives.

Chapter 3 covers the basics of practical 80386 assembly language programming. It starts with simple programs and proceeds through bit manipulation, shifting, decision making, array manipulation, table lookup, character manipulation, code conversion, multiple-precision arithmetic, and data structure manipulation. An example section contains complete programs. Final sections discuss parameter passing methods, ways to make programs run faster, and common programming errors.

Chapter 4 describes I/O. It covers I/O addressing, I/O instructions, programmable I/O chips, interrupts, and direct memory access (DMA). It includes discussions of the popular 8250 serial interface, 8255 parallel interface, 8259 interrupt controller, and 8237 DMA controller.

Chapter 5 deals with the 80386's memory management facilities. It first describes the processor's operating modes. It then covers segmentation, paging, memory protection, the creation of descriptors, privileged instructions, and the initialization of memory management systems.

Chapter 6 explains the 80386's task management features. It discusses task descriptors, task state segments, privilege levels, task switching, task linking, and task address spaces. It also covers I/O privilege levels, I/O permission bit maps, and the initialization of tasking systems.

Chapter 7 deals with the 80386's exceptions and debugging features. It describes the sources of exceptions, error codes, and exception conditions. A final section discusses the debug registers.

Chapter 8 presents the 80386's hardware features. It includes a general overview of the signal structure and bus operations. It also covers numerical coprocessors, memory interfacing, and cache memory.

The appendixes contain the instruction sets of the 80386 and the 80287 and 80387 numerical coprocessors. They also contain summaries of the status flags and descriptor formats, as well as summaries of the differences between the 80386 and earlier processors.

This book should give the reader enough understanding of the 80386 processor to tackle many projects. These could include the design and development of embedded systems, the rewriting of programs for 80386-based computers, and the creation of new utilities for the 80386. This book should also alert readers to likely new features in 80386-based computers and systems. It should thus help readers prepare for the inevitable transition from 16-bit processors to 32-bit processors in both personal computers and embedded systems.

Many people and organizations contributed to the writing of this book. Steve Guty of Bantam Books was the acquisitions editor. Claudette Moore, Gary Masters, and

Preface

Mike Halvorson of Microsoft Press provided many editorial suggestions. Phil Barrett, Paul Butzi, and (especially) Hans Spiller of Microsoft were willing to share their experiences in programming the 80386. George Fahouris, Lisa Figlioli, Clif Purkiser, and Doug Rick of Intel provided materials and encouragement. Tracey McAllister of Intel, Rosemary Morrissey of IBM, and others provided photographs of actual systems. Microsoft and Compaq Computers loaned me equipment and manuals. I also profited from conversations with Chris Ruff of Sorrento Valley Associates, Irvin Stafford of Unisys Corporation, and Dave Flower of Compaq Computer. Two anonymous reviewers offered many last-minute additions, clarifications, and corrections. After a brief siege of grumbling, I made most of the changes. I certainly appreciate their efforts, particularly the Bantam reviewer's. Last but not least, I want to acknowledge the support and encouragement of my wife, Donna, and my daughters, Elizabeth and Stacy.

This book is dedicated to two of my teachers at Roosevelt High School in Seattle, Washington: Elizabeth Clark and Ann Grodal. They showed me the importance of words and ideas and challenged me to strive for clarity and insight.

C • H • A • P • T • E • R

1

Introducing the 80386

*Why, a four-year old child
could understand this report.*

Run out and get me a four-year old child.

I can't make head or tail out of it.

Groucho Marx
"Duck Soup"

The 80386 microprocessor is a major advance over the 8088, 8086, and 80286 devices. Its 32-bit architecture allows it to handle twice as much data at a time as can 16-bit processors. Computers based on it are thus much more powerful than such systems as the IBM PC and PC clones (based on the 8088), the AT&T 6300 and IBM Personal System/2 Model 30 (based on the 8086), and the IBM PC AT, AT clones, and the IBM Personal System/2 Models 50 and 60 (based on the 80286). Yet the 80386 can run MS-DOS programs and all others written for those earlier processors. The 80386 thus brings the power of a superminicomputer (such as the DEC VAX line) or a mainframe (such as IBM's 360/370 lines) to the chip level. An 80386-based computer, such as the IBM Personal System/2 Model 80 (Figure 1-1), can do more than computers of the 1970s that cost hundreds of thousands or even millions of dollars.



Figure 1-1

The IBM Personal System 2/Model 80, an 80386-based computer.
Photo courtesy of IBM Corporation, Information Systems Group,
Rye Brook, New York.

KEY FEATURES

The 80386 is more than just an expanded 8088 or 80286. Among its key features are:

- Memory capacity of 4 gigabytes (Gb or G). A gigabyte is 1,024 megabytes (approximately 1 billion bytes; see Table 1-1). This is 256 times the capacity of the 80286 and 4096 times that of the 8088. Figure 1-2 shows how the memory capacity of Intel processors has increased with time. The slope of the curve is impressive, even with a logarithmic vertical scale. The 8080's 64K capacity is just barely above the horizontal axis.

Table 1-1
Memory Units and Their Meanings

Unit	Number of Bytes	Prefix Meaning
Kilobyte (Kb)	1,024	Thousand
Megabyte (Mb)	1,048,576	Million
Gigabyte (Gb)	1,073,741,824	Billion
Terabyte (Tb)	1,099,511,627,776	Trillion

Table 1-2
Four Gigabytes of Memory in Various Units

Unit	Number Required
2-Mb board	2,048
640-Kb board	6,450
256-Kb board	16,384
1-megabit chips	32,768
256-kilobit chips	131,072

Four gigabytes is enough memory to remember the entire U.S. national debt, although not enough to do anything about it. Besides, gigabyte is a new word with a great sound. It is a major addition to one's technical vocabulary, now that most people know about kilobytes and megabytes.

Of course, although gigabytes sound good, they are not terribly practical at present. Four gigabytes would occupy 2000 2-Mb boards or over 6000 640K boards (see Table 1-2). To house that much memory, your computer would need a huge chassis, thousands of empty slots, and a gargantuan power supply! But some ancient pioneers may remember back to the bygone 1970s when megabytes were equally impractical, and 64K was more memory than any reasonable person would ever need. For those who like to look ahead, the next stage (see Table 1-1) is terabytes.

- Ability to run 8086 programs in their own environments. The 80386 can switch back and forth between an 8086 mode and its own native operating mode. An 80386-based computer can thus run popular MS-DOS

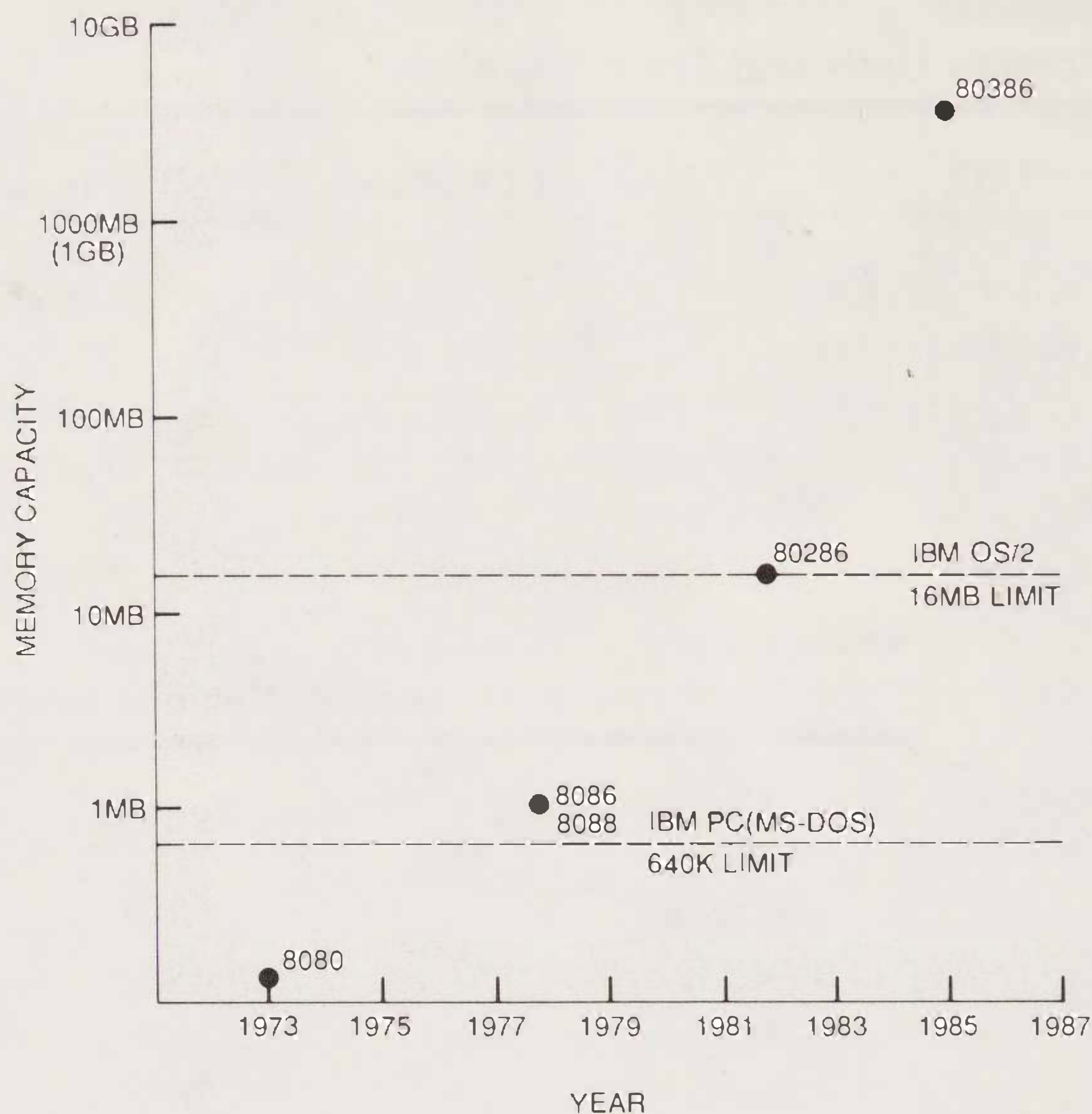


Figure 1-2

Processor memory capacity vs. time for Intel CPUs.

software without giving up access to more advanced features. This is a big improvement over the 80286 processor, which cannot run MS-DOS programs in its native (protected) operating mode. Many 80286 features, such as its 16 Mb of addressable memory, are therefore inaccessible to MS-DOS users. OS/2, which runs on 80286- and 80386-based computers, gives access to 80286 features but not to those new to the 80386.

- Support for operating systems that run several tasks at once (called *multitasking*). The 80286 and 80386 both have special instructions, data structures, and other features intended for such systems. Multitasking is important in personal computers and in such applications as computer-aided design (CAD), computer-aided manufacturing (CAM), computer-aided engineering (CAE), robotics, artificial intelligence, industrial

control, process control, military and aerospace systems, instrumentation, and workstations.

- Ability to address single units or blocks of memory as large as 4 Gb. We call such addressable units of program or data memory *segments*. On the 80386, even large programs can fit in a single segment. This avoids the extra programming and machine time required to check for segment boundaries and move from one segment to another.
- Support for virtual memory. That is, the 80386 can readily address more memory than is physically present. The operating system can move data and programs to and from disk as needed. The programmer does not have to manage physical memory systems that may vary with computer model or be expanded as time passes.
- Special on-chip hardware for shifting, multiplication and division, and address generation. This new hardware makes instructions execute faster. For example, a device called a *barrel shifter* can shift up to 64 bit positions in a single clock cycle. Thus all shifts take the same amount of time, regardless of the number of positions involved. Long shifts are common in graphics, communications, compiling, image processing, and string processing. The overlapped execution of address calculations (such as indexing) is another major reason for the 80386's improved performance. Even the most complex addressing mode now takes only one extra clock cycle, as compared to up to twelve on earlier processors.

The 80386 has other new features as well. For instance, it has debug registers for use in program development. It also has instructions for bit manipulation, bounds checking, and module handling. Improved hardware and instruction facilities greatly speed up arithmetic operations, shifts, and addressing. Externally, the 80386 has a new, more powerful 32-bit floating point unit, the 80387 numeric data processor (see Appendixes G and I).

The 80386 has a more generalized architecture than its predecessors. A frequent criticism of the 8086 and 80286 processors is their dedication of many registers to specific purposes, such as accumulators, indexes, and base addresses. The 80386 allows users to ignore much of this specialization, although they can take advantage of it if they wish. More consistency is particularly helpful to compiler writers, whose programs must translate high-level computer languages into low-level executable code (*machine language*). The more consistent the architecture, the easier it is for a compiler to do automatic translation without any special analysis or concern for context.

VIRTUAL 8086 MODE

A major improvement in the 80386 is its new virtual 8086 (V86) mode. This mode gives the 80386 its ability to run both 8086 programs and new programs requiring advanced features. Like the 80286, the 80386 has two main operating modes:

Real mode, in which it acts like an 8086. That is, it behaves like a 16-bit processor with access to 1 Mb of memory.

Protected (virtual) mode, in which it has access to all its features and facilities. That is, it behaves like a 32-bit processor with access to 4 Gb of memory.

The problem is the difficulty of switching between real and protected modes. The two conflict, and only one can be active at a time. Unfortunately, as we mentioned, MS-DOS does not run in the protected mode on the 80286 (or the 80386). Thus we must give up either MS-DOS software or protected mode features.

The 80386 adds an intermediate stage called *virtual 8086 mode*. It is an option within the protected mode, selected by a flag. Virtual 8086 mode does not conflict with the full protected mode. In it, the processor acts like an 8086. However, some 80386 features remain active, under the control of a special program called a *virtual 8086 monitor*. The result is that 8086 programs can coexist with programs using advanced 80386 features and running in protected mode. Virtual 8086 mode thus provides a bridge between MS-DOS software and new 80386 features.

In fact, in the virtual mode, an 80386 can behave like several 8086s, each with its own operating system, programs, and memory areas. Users may think they have 8086s for their own private use, even though they are actually sharing an 80386-based machine. We call such a simulation a *virtual machine*.

A key point here is that in V86 mode, the 80386 executes programs just like an 8086. It does not emulate the 8086. That is, it does not translate 8086 instructions into its own native mode. Emulation would make an 80386 run slower than an 8086. In fact, the 80386 runs faster because of its higher clock speed, more extensive pipelining, and extra arithmetic hardware.

PIPELINED ARCHITECTURE

Figure 1-3 shows the functional units inside the 80386 microprocessor. The key point is that all units operate at the same time. That is, they do their jobs simultaneously on different operands. We call this approach *pipelining*, since it works like a pipeline that moves things continuously rather than in discrete units.

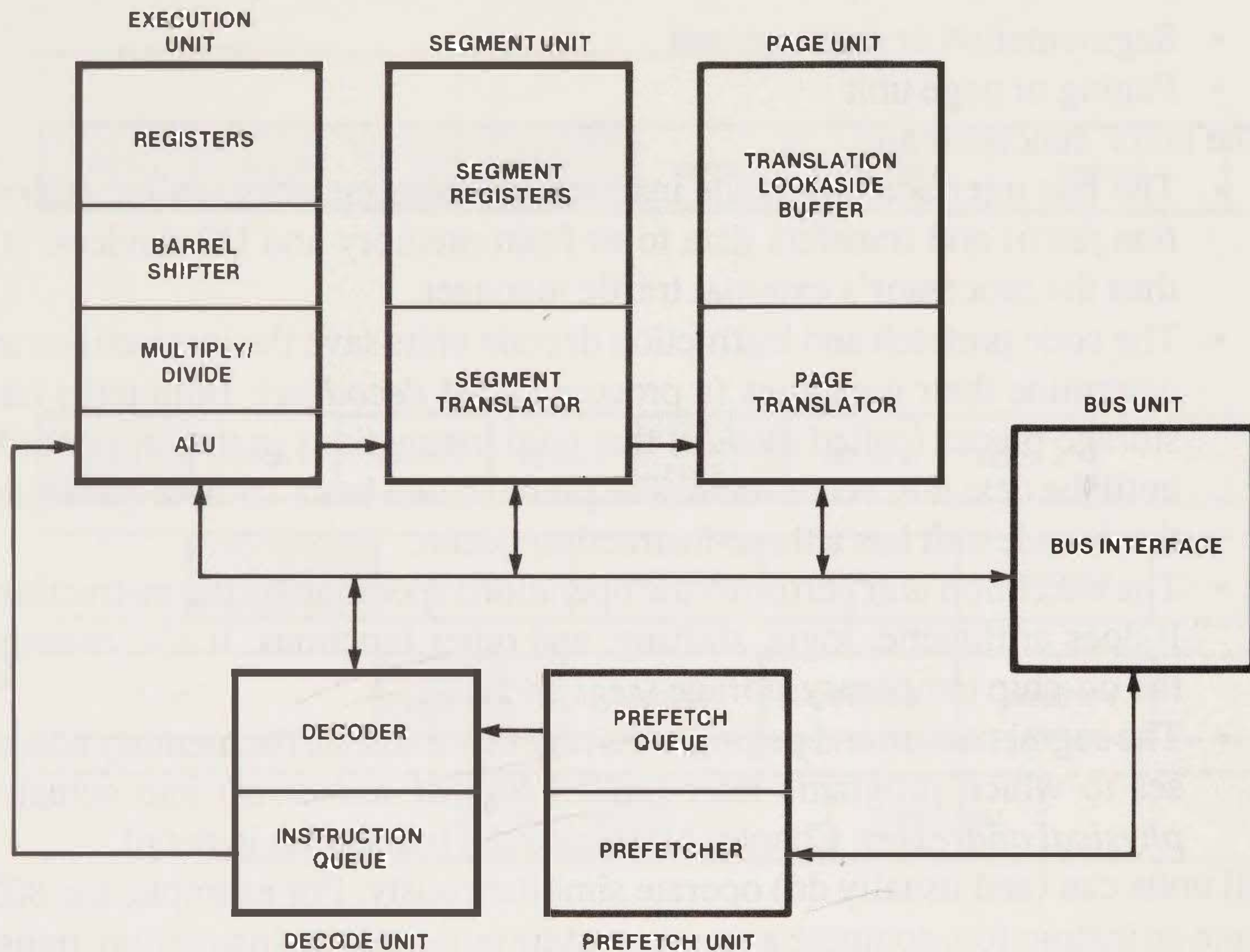


Figure 1-3
80386 functional units.

You may compare a computer pipeline with an automobile assembly line. The line does not finish one car and then start another. Instead, it works on many cars at once. One car may be having its doors welded, another its motor inserted, a third its windows attached, and a fourth its paint applied, all at the same time. Clearly, production is much higher if several workstations operate simultaneously.

An 80386 has six distinct units or workstations, as Figure 1-3 shows. They are, starting at the far right and moving clockwise:

- Bus interface unit
- Code prefetch unit
- Instruction decode unit
- (Instruction) execution unit

- Segmentation or segment unit
- Paging or page unit

The units' functions are:

- The bus interface unit reads instructions from memory (called *instruction fetch*) and transfers data to or from memory and I/O devices. It is thus the processor's external traffic manager.
- The code prefetch and instruction decode units save the instructions and determine their meanings (a process called *decoding*). Both units have storage places (called *queues*) that hold instructions in the proper order until the next unit needs them. The prefetch unit has a 16-byte queue, and the decode unit has a three-instruction queue.
- The execution unit performs the operations specified by the instructions. It does arithmetic, logic, shifting, and other functions. It also manages the on-chip temporary storage (*registers*).
- The segmentation and paging units together translate the memory addresses to which programs refer (called *logical addresses*) into actual or *physical addresses*. Chapter 5 explains the translation in detail.

All units can (and usually do) operate simultaneously. For example, the 80386 can execute an instruction, compute a memory address, decode an instruction, transfer data to or from memory, and do other jobs at one time. Of course, like the automobile assembly line, the 80386's pipeline works at full speed only if all inputs are available when units need them.

At top speed, the 80386 is working on several instructions at the same time. Then it need not wait for one part of an operation to finish before starting another. Of course, it takes a while to fill the pipeline at first.

Figure 1-4 compares pipelined operation to the sequential operation of early processors such as the Intel 8080. The leftmost part of the figure shows how the pipeline fills initially. Instruction 1 works its way through the empty units much as it does in the sequential processor at the top. However, while the pipelined processor is executing instruction 1, it is also fetching instructions 3 and 4 and decoding instruction 2. At this point, the pipeline is full, and all units are busy. By the end of the interval, the pipelined processor has executed four instructions as compared to the sequential processor's two. Furthermore, the pipelined processor has fetched instructions 5 and 6 and decoded instruction 5.

A pipelined processor requires special programming techniques for optimal performance. Slowdowns occur whenever the pipeline must be filled. The usual reason is a transfer of control that forces the processor to clear its queues and start executing in-

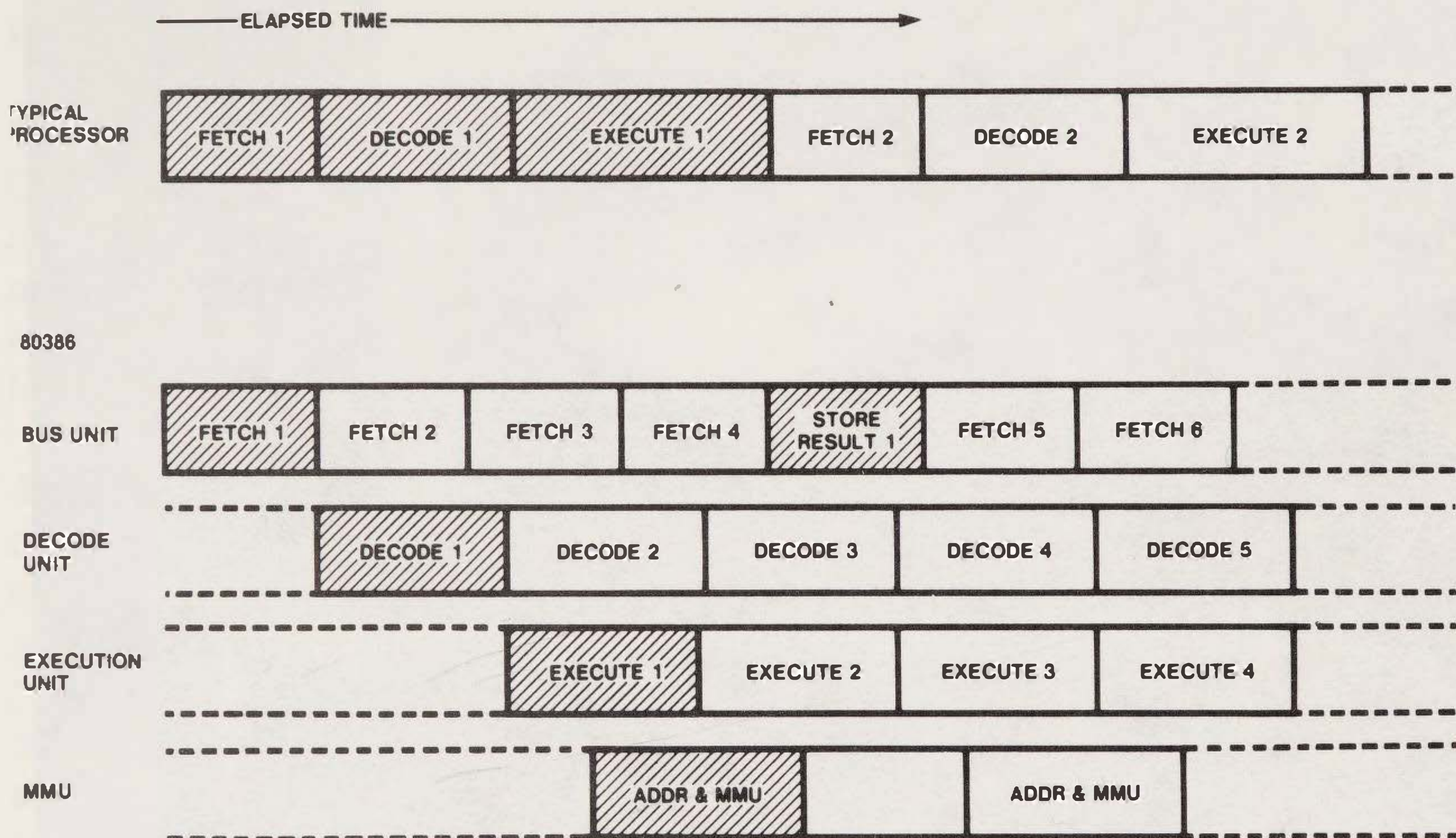


Figure 1-4
Instruction pipelining.

structions somewhere else in memory. The programmer must minimize the number of jumps to keep this from happening too often. Software theorists will be glad to hear that the 80386 makes GOTOs highly undesirable. They are not only harmful to program structure, but they also slow down the hardware.

TYPICAL APPLICATIONS

What applications need the 80386's capabilities? Leading areas include:

- Personal computers
- CAD/CAM/CAE (computer-aided-design/manufacturing/engineering) systems
- Robotics
- Artificial intelligence

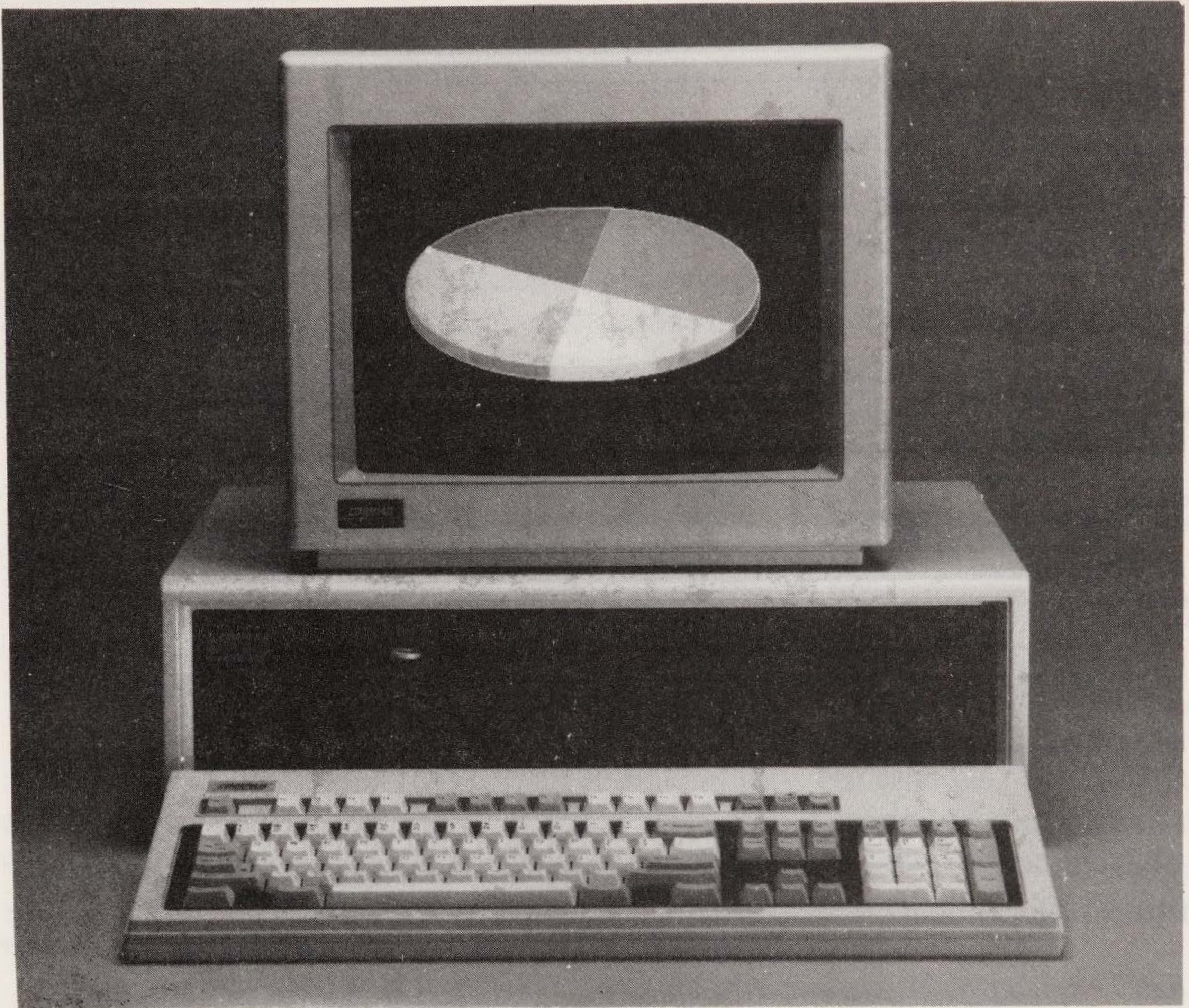


Figure 1-5

The Compaq Deskpro 386. Photo courtesy of Compaq Computer Corporation, Houston, Texas.

- Signal processing

Other 80386 applications include laser printers, graphics, desktop publishing, process control, industrial control, test equipment, instrumentation, communications equipment (such as network controllers or servers), office automation, banking terminals, guidance and control, and military systems. Let us briefly discuss why such applications need the features and power of an 80386.

Personal Computers

There are several markets for more capable personal computers such as the Compaq Deskpro 386 shown in Figure 1-5. One market is the so-called power users. They are people whose documents, spreadsheets, databases, or other applications require more memory and processing power than is currently available. In fact, they need more of everything, regardless of what is available. And they will always need more, regardless of what advances occur.

Power users include writers who want to check spelling and grammar, produce indexes and tables of contents, and process documents that are only slightly longer than the *Encyclopedia Britannica*. Who reads the output? Probably no one, but it keeps editors, typists, typesetters, printers, and librarians busy. It also fills government repositories and satisfies contracts and regulations.

Other power users are financial analysts whose models cover thousands of cells, compute complex formulas, and combine data from hundreds of sources. The result is an almost infinite number of factors that you can adjust to get the results you already know are correct. Still other power users have huge databases, large projects to manage, complicated graphics to create, or complex functions (such as product mixes or transportation costs) to optimize.

Another market consists of scientists, engineers, and programmers who want to use personal computers as low-cost workstations. Solving huge sets of differential equations, statistical analysis, simulation of complex phenomena, and compiling long programs written in high-level languages are typical tasks requiring large amounts of computing power. The needs are particularly great if the user wants interactive operation or an intelligent interface, online help, macro or batch file capabilities, extensive graphics, and databases of common procedures and results.

Such workstations are useful even when the overall problem requires a supercomputer to solve. Using a workstation for program development, data entry, and analysis of results reduces the burden on the large machine. It also results in lower cost and greater convenience.

Still another market consists of people who want low-cost multiuser capabilities. Figures 1-6 and 1-7 show 80386-based multiuser systems from Integrated Business Computers and Prime Computer, respectively. Owners of such systems may want one terminal for data entry and one for billing. Or they may want several terminals for order entry, interviewing, information retrieval, or financial reporting. The key is that all users must share a common database, so several personal computers will not do the job. We might call this *subdepartmental computing*, since it involves units smaller than

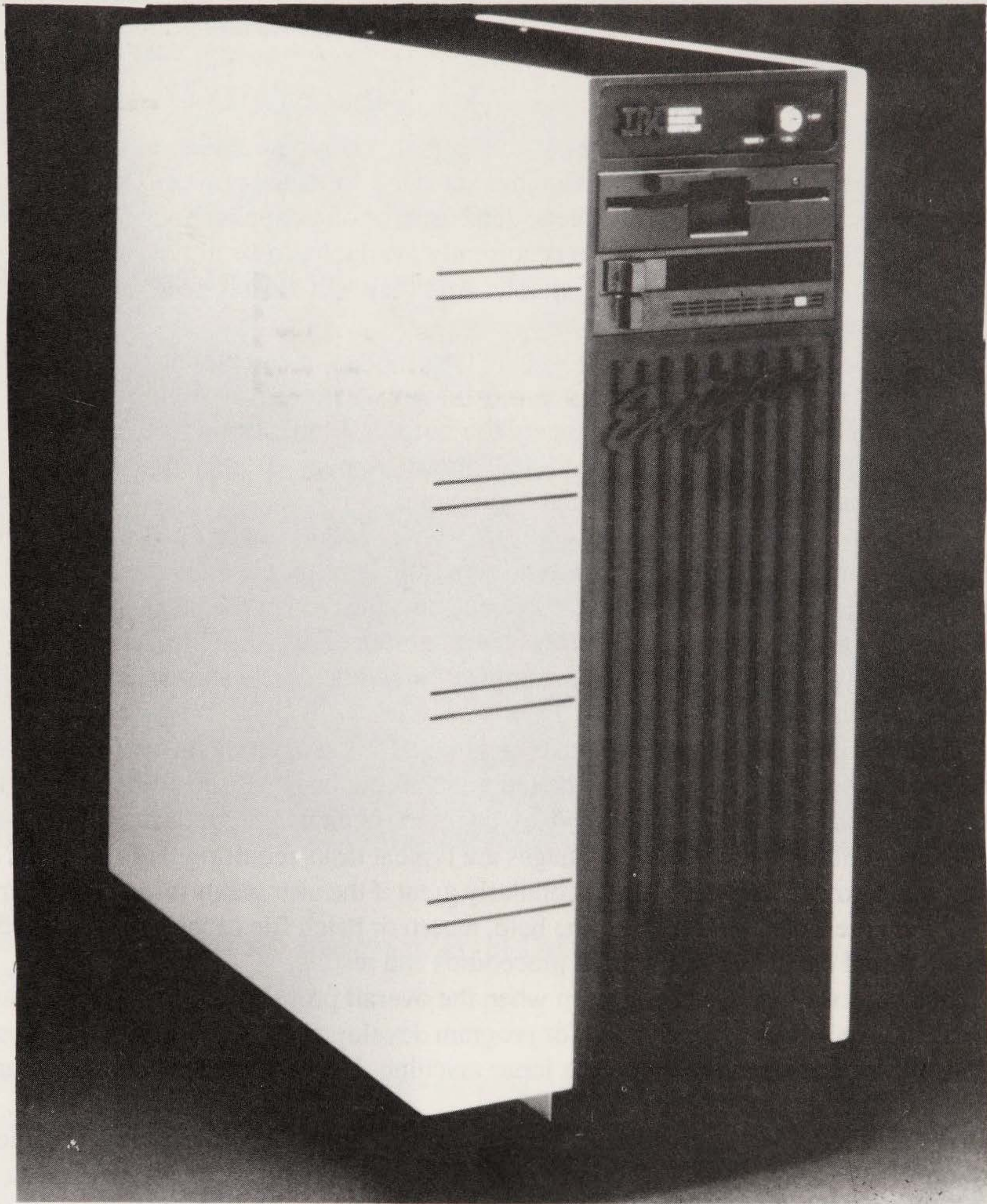


Figure 1-6
The Integrated Business Computers Ensign 386:100 multiuser computer.

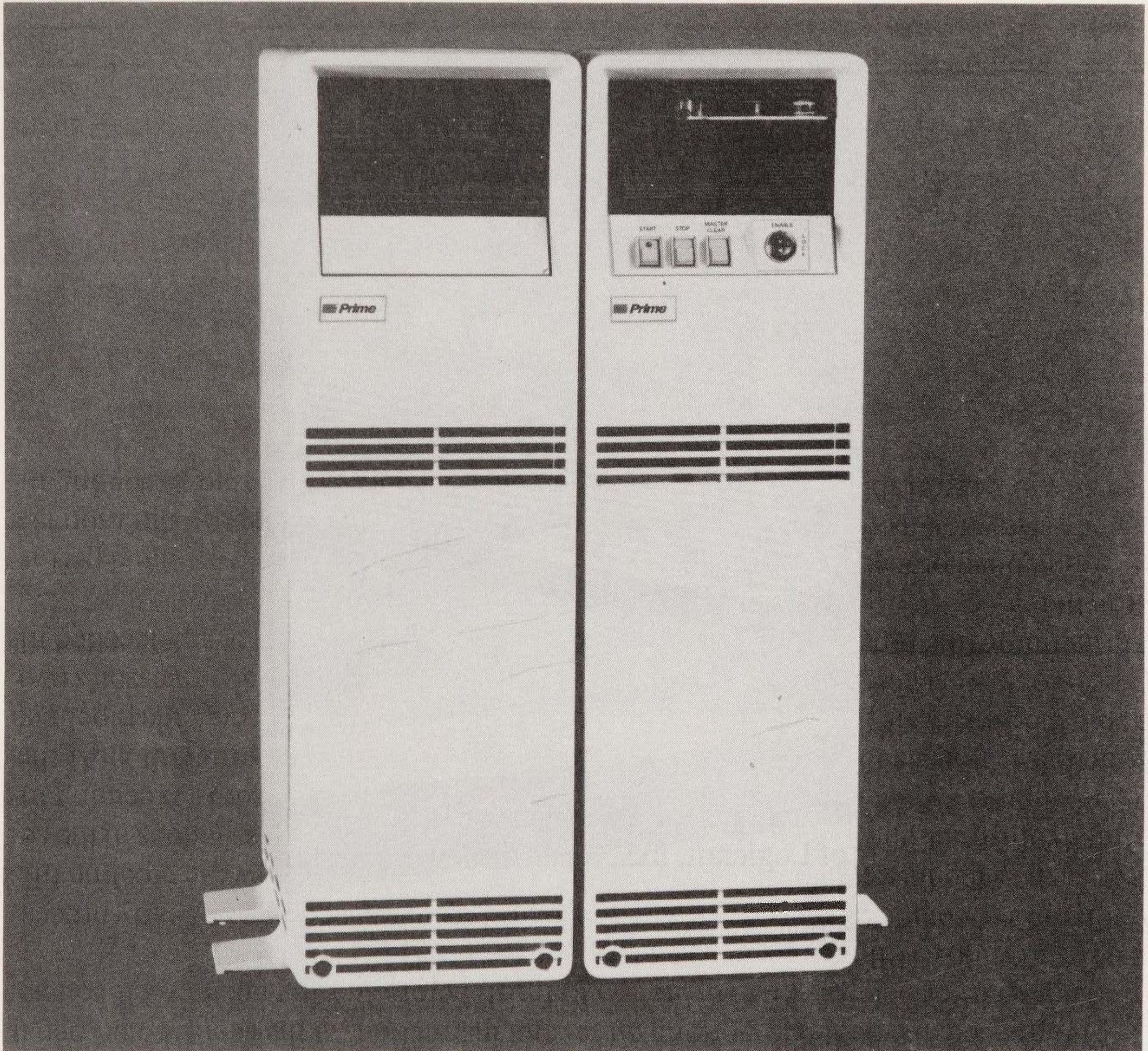


Figure 1-7

The Prime EXL 316 supernmicrocomputer. Photo courtesy of Prime Computer, Inc., Natick, Massachusetts.

the departments handled by minicomputers. A doctor's office, a pharmacy, a realtor's office, or a restaurant would be typical applications.

An 80386-based computer can also provide MS-DOS capabilities to users of other machines. For example, Logicaft's 386WARE (Figure 1-8) allows DEC VAX users to run MS-DOS programs from their terminals. Here the 80386-based machine is both a file server and a working computer.

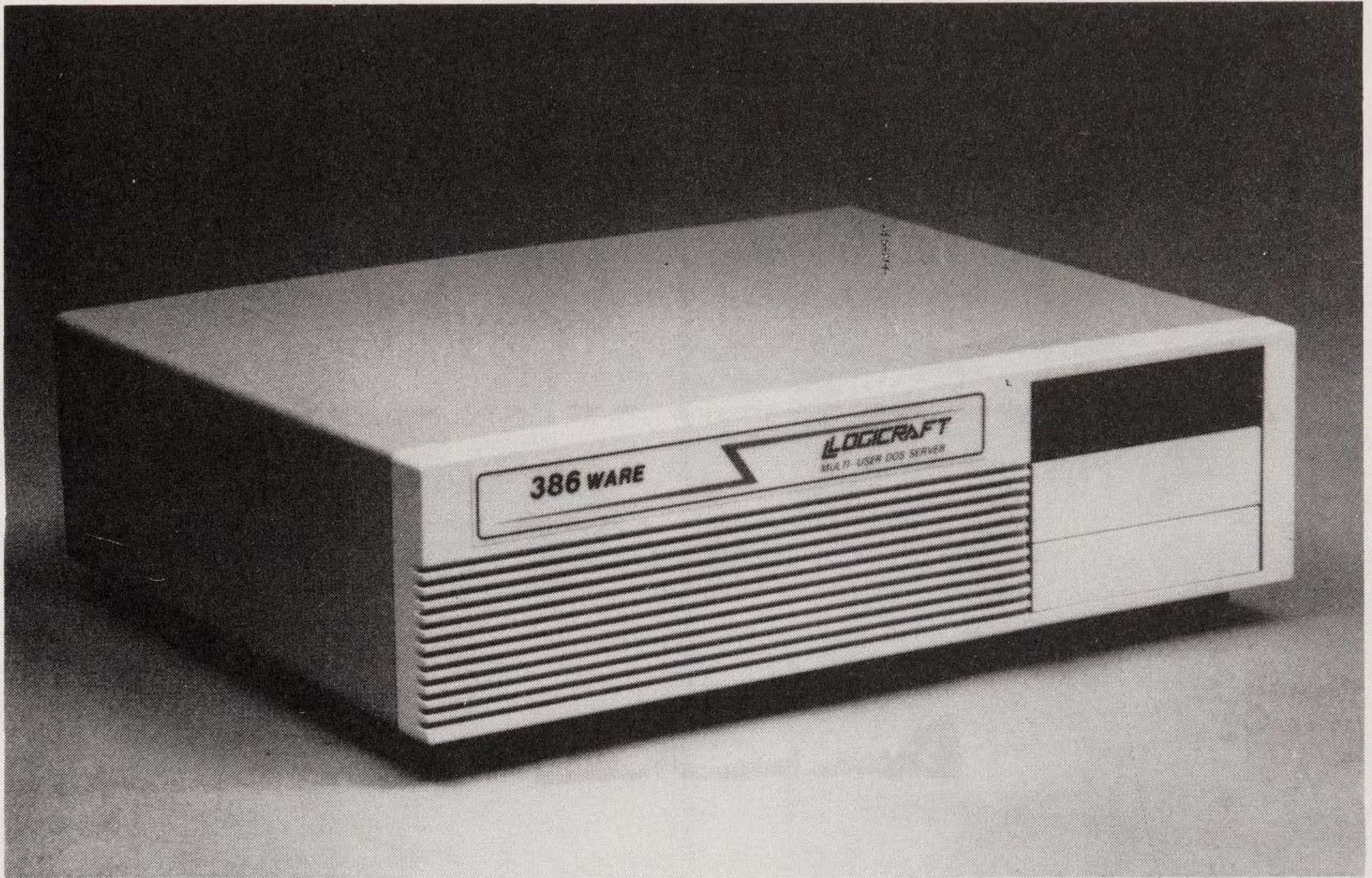


Figure 1-8

The Logiccraft 386WARE multiuser DOS server for DEC VAX networks. Photo courtesy of Logiccraft, Inc., Nashua, New Hampshire.

CAD/CAM/CAE Systems

CAD/CAM/CAE systems also require large amounts of processing power. A typical application is the Daisy Systems Personal Logician 386 shown in Figure 1-9. Although the name makes it sound like the perfect tool for Aristotle's students, it is actually an 80386-based engineering workstation used to design, simulate, and test integrated circuits. Figure 1-10 shows another typical application. Here an architect is using an 80386-based personal computer to design a building. CAD/CAM workstations like these must be able to:

- Display complex objects. Many systems must do tasks such as rotation, shading, mirroring, scaling, and zoom.
- Manipulate large libraries of standard parts, drawings, images, shapes, waveforms, and procedures.



Figure 1-9

The Daisy Systems Personal Logician 386. Photo courtesy of Daisy Systems Corporation, Mountain View, California.

- Capture data from documents and other sources. For example, a CAD/CAM system may have to scan artwork masters and convert them to standard formats for processing.



Figure 1-10

An IBM Personal System/2 Model 50 computer used to plot a subdivision. Photo courtesy of IBM Corporation, Information Systems Group, Rye Brook, New York.

- Perform complex procedures (*algorithms*) such as routing, curve fitting, optimization, worst-case analysis, filtering, and reliability or stability analysis.
- Provide the ability to easily vary parameters for rapid “what-if” analysis. For example, suppose a planner was using a computer as shown in Figure 1-10 to plot a subdivision. He or she might want to try several alterna-

tives and compare their costs, profits, environmental effects, and other features.

- Simulate systems to estimate their performance and see how they work under stress, transients, or failure modes.
- Create test programs and patterns.
- Allow users to easily manipulate complex objects, graphics, and systems.
- Generate parts or wire lists, schematics, blueprints, bills of material, database entries, flowcharts, and other documentation.
- Handle specialized design languages, databases, and other packages.
- Provide familiar interfaces such as those of common test instruments, popular programs, or manual systems.

Typical application areas include vehicles, electronic circuits, assembly lines, computer and communications networks, mechanisms, buildings and other structures, processing plants, chemistry, microbiology, and genetic engineering. CAD/CAM systems can also aid surgeons, pharmacists, urban planners, geologists, power engineers, motion picture makers, aircraft designers, and civil engineers. Here again, extensive graphics (particularly three-dimensional color), user interaction, and checking of inputs and results add to the computational load.

An 80386-based CAD/CAM/CAE workstation (such as the ones shown in Figures 1-1, 1-9, and 1-10) can provide extensive capabilities at low cost. Besides, it can also run popular word processing, spreadsheet, and database management software. The ability to run multiple operating systems is important here, since design tools often run under Unix, whereas personal computer programs run under MS-DOS. For example, Daisy Systems' Personal Logician 386 (Figure 1-9) runs MS-DOS as a task in a modified Unix environment. Design tools can then use Unix's 16-Mb address space instead of being restricted to MS-DOS' 640K.

Robotics

Robotics also requires a large amount of low-cost processing power. The more processing power a robot has, the more extensively it can analyze its surroundings and predict the consequences of its actions. If a robot is to recognize objects by vision or touch, move in a crowded environment containing obstacles, understand natural language commands, or do complex operations, it must have a substantial computer. It may also have to remember a large amount of data and many procedures, combine data from

many sources, and learn from what it does. The more processing power it has, furthermore, the more the robot can do on its own without constant supervision by an operator.

Other things that a smarter robot can do include:

- Communicate with other robots and coordinate activities with them
- Perform self-test, self-checking, and self-diagnosis procedures
- Provide local facilities for modifying and developing programs
- Perform movements involving more degrees of freedom, closer tolerance, and higher resolution
- Keep detailed records of the results of previous operations
- Recover from errors and system failures
- Use radar and sonar for navigation

Artificial Intelligence

Artificial intelligence (AI) is still another application area requiring extensive processing power. The more processing power a computer has, the better it is at such AI tasks as:

- Evaluating the many alternatives involved in expert systems
- Implementing such techniques as inference, commonsense reasoning, backward chaining, and goal seeking
- Performing complex procedures involving backtracking, extensive decision logic, and many relationships
- Compiling programs written in AI languages such as LISP and Prolog, as well as in specialized high-level packages
- Handling uncertainty in data, procedures, and requests for information

A more powerful computer can also access larger knowledge databases more quickly, provide more extensive tools, and consider a wider range of alternatives. It may even be able to respond to simple situations in real time. Typical application areas include medical diagnosis, financial planning, vehicle repair, program interfaces, circuit and chip design, weather forecasting, automatic programming, genetic engineering, planning systems, and robotics.

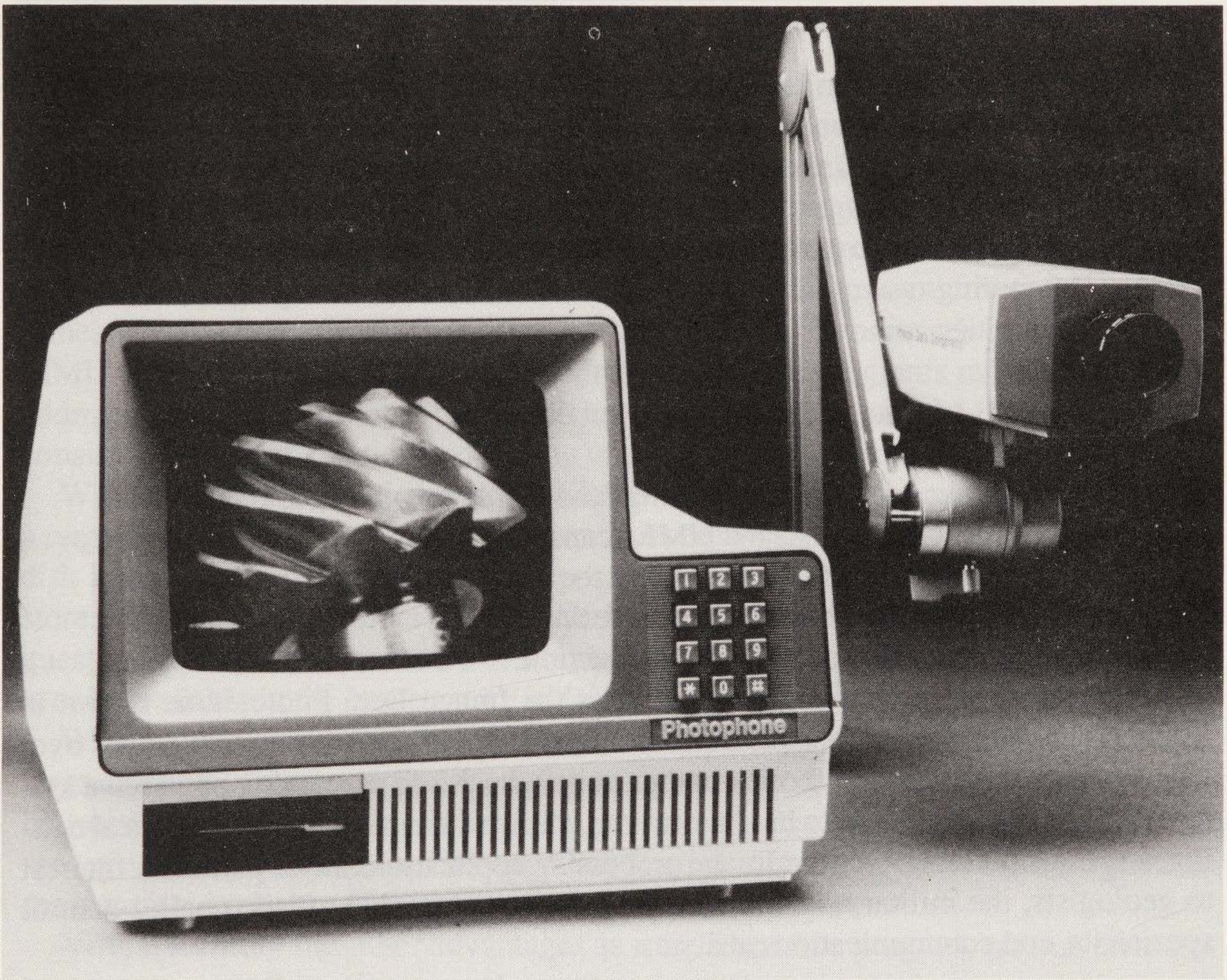


Figure 1-11

The Image Data Photophone, a device for sending still images on telephone lines. Photo courtesy of Image Data Corporation, San Antonio, Texas.

Signal Processing

Signal processing has long been an application requiring a lot of computing power. Typical uses include:

- Enhancing images derived from satellites, remote cameras, or underwater photography. Is that dark spot in the field a scarecrow or a missile? Military intelligence surely wants to know.
- Interpreting data received from radar and sonar systems. For example, fighter pilots want to know quickly whether an object is an enemy aircraft

or a large bird. Similarly, ship captains want to quickly distinguish an Exocet missile from an oil tanker or a great white shark.

- Filtering of noise from images and communications channels.
- Compressing images for efficient storage and transmission. Both time and cost are important factors here.
- Comparing waveforms for speaker identification, recognition of irregularities, removal of known or baseband sources, or isolation of changes.
- Analysis of visual data for use by robots.
- Speech recognition and synthesis.
- Interpreting, storing, transmitting, and displaying medical images such as X-rays, CAT scans, and NMR scans.
- Vibration analysis for determining the failure modes of structures, such as buildings, bridges, and nuclear reactors.
- Music or sound synthesis and production.

A typical image processing application is the Image Data Photophone shown in Figure 1-11. This system accepts input from a camera and sends the still images over a telephone line. It can also receive and print images. More processing power in a system like this would allow it to hold more data, enhance images, provide graphical editing, and transmit images faster. Image processing applications are of particular interest to geologists, the military, health care professionals, graphic artists, quality control specialists, and communications analysts.

KEY CONCEPTS

We must understand the following new concepts to appreciate the 80386:

- Virtual memory
- Multitasking
- Multiuser systems
- High-level language and operating-system-oriented machines

While these ideas are new to microprocessors and personal computers, mainframes, minicomputers, and specialized computers have used them for many years. Many of the 80386's features are clearly derived from larger machines.

Virtual Memory

Virtual memory refers to systems in which the combined size of the program and data areas may exceed the physical memory. The operating system keeps the currently used parts of the program and data in memory. It saves the rest on disk until it is needed. Before the computer accesses a location, a memory management unit (MMU) determines whether it is actually in memory. If it is, the transfer proceeds as usual. All the MMU must do is convert the logical addresses to which programs refer into physical addresses that identify actual memory locations. The 80386 has its MMU on board. It consists of the segmentation and paging units shown in Figure 1-3.

What happens if a location is not available? Then the MMU reports an error (called a *page fault*). The operating system must take control and load the required area from disk into memory. The virtual memory system thus acts like a clerk in a department store. He or she fills an order from the shelves if possible. If not, the next step is to request the items from a stockroom or warehouse. The size and style you want, of course, is only available in the branch store in Outer Mongolia.

Why would you want virtual memory? First, it lets us write programs that refer to large amounts of memory. The operating system can handle the details of moving areas between memory and disk. As far as the programmer is concerned, memory and disk are a single continuous unit. This approach is much simpler than having a program do the transfers explicitly (a method called *overlays*).

Virtual memory has other advantages as well. The same program can run on many computers, regardless of how much memory they have or how it is arranged. You need not revise a program to use larger memory units, less expensive memory, or a new computer model. Virtual memory also simplifies multiuser systems. Each user can refer to the entire memory. The memory management unit isolates users by keeping each one's address space (program and data areas) separate.

There are two major ways to implement virtual memory. One is by dividing program and data areas into logical units called *segments*. Different segments can then go into different areas of virtual memory. Segmentation is particularly convenient for operating systems, since they generally must assign areas for each program and its data anyway. However, segmentation is a nuisance to programmers, since they must divide their code and worry about intersegment transfers and segment boundaries. The overhead also makes programs run more slowly. The 80386 uses segmentation but allows large enough segments so that most programs need not struggle with it. Note that the sizes and numbers of segments depend on individual programs, not on the underlying physical memory.

Table 1-3
Page Table for the Example Memory System
(Figure 1-12)

Frame Number	Page Number
1	2
2	13
3	27
4	6
5	38
6	51
7	18
8	5

The second approach is *paging*. Here, the operating system divides memory into areas of fixed size called *pages*. Disk operations transfer one page at a time. The memory management unit must only determine whether a particular page is currently in memory. If not, the unit must fetch it from disk. Unlike segmentation, paging is invisible to the programmer. Programs and data can extend over many pages with no problems. The 80386 allows paging but does not require its use.

How does paging work in practice? Figure 1-12 shows a simple example of a paged virtual memory system. The addressable memory (from a program's point of view) consists of 64 pages. However, the physical memory consists of only eight page-sized units or *frames*. Think of frames as slots into which we can put a page of virtual memory (or address space). Table 1-3 lists which pages currently occupy the frames.

It is quite simple to access a page that is in memory. For example, suppose that a program's next instruction is on page 2; fetching it is straightforward. Page 2 is in memory, and all the processor must do is find it (through a procedure called *address translation*). This involves looking up the page number in Table 1-3 (the *page table*) and reading the corresponding frame number (1 in the current case).

Note that the pages are not ordered. The processor must use the page table to determine whether a page is in memory and, if it is, which frame it is currently occupying.

What happens if the next program instruction is on page 33? Now we have a problem, as page 33 is not in memory. A page fault therefore occurs. The operating system then takes control and must do the following:

1. Make room for page 33 in memory by removing a page (say, 38) and saving it on disk.
2. Load page 33 into memory in the frame (5) formerly occupied by page 38.
3. Change the page table to indicate that frame 5 now contains page 33, not page 38.
4. Return control to the program that caused the fault.

Obviously, recovery from a page fault is a complex process that may take a long time.

The situation may be even more involved. A page fault may occur in the middle of an instruction. For example, an add instruction may require data from memory. If obtaining the data causes a fault, the processor must suspend the add instruction while the operating system fetches the required page. The processor must then either start the instruction all over again (called *restart*) or resume it. Resuming an instruction requires extra storage and logic, since the processor may have to save many intermediate results.

Note that one instruction could cause several page faults. For example, suppose it moves data from one memory location to another. The original instruction fetch could cause a fault, and each data transfer could cause another. The 80386 can restart all instructions from any point in their history at which a page fault could occur. However, it cannot resume instructions.

Although virtual memory has advantages, it also introduces new problems. Clearly, moving pages to and from memory (a process called *swapping*) takes time. A program will run slowly if it causes many page faults. We say that such a program (rather descriptively) is *thrashing*. The problem is clearly the same as maintaining shelf inventory in a store. You want to use your limited shelf space wisely to minimize the number of trips back to the stockroom. The usual solution, of course, is to keep the most popular sizes and styles immediately available.

How do we use a limited memory space wisely? The common method is to only load a new page when required. We call this a *demand paged system*. Of course, to make room for a new page, the system must store an old one on disk. A common approach is to remove the page that has gone the longest time without being used. We call this approach a *least recently used* (LRU) algorithm. To implement it, the operating system must have a way of measuring page usage over time.

In practice, most programs refer to a few pages repeatedly (called a *working set*). Once the system has brought a program's working set into memory, it should execute without further swaps. For example, a program using the memory system in Figure 1-12 might have its main code on pages 2, 5, and 6 and its data on page 51. As long as it did not refer to other pages, no swaps would be necessary to execute it.

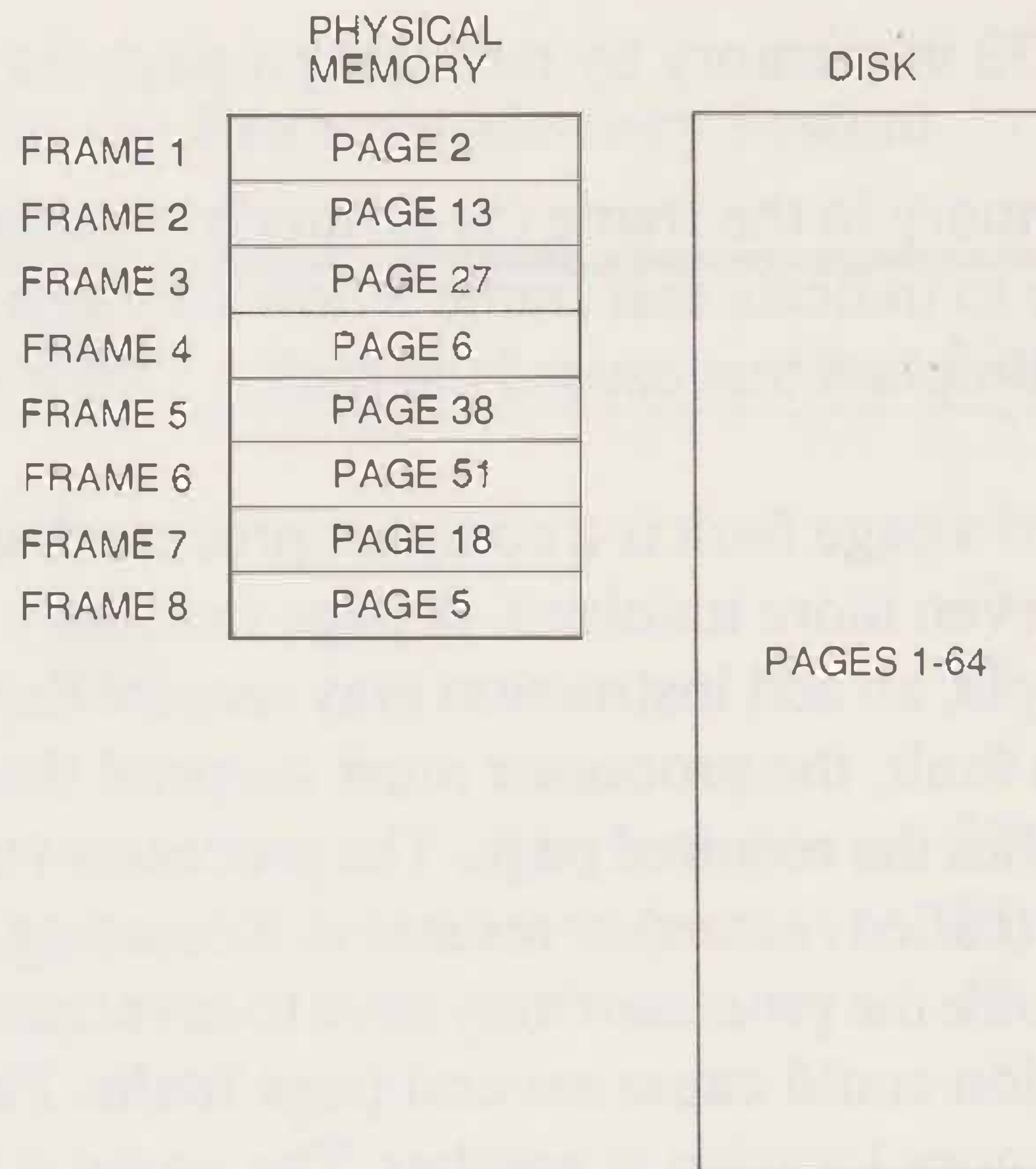


Figure 1-12

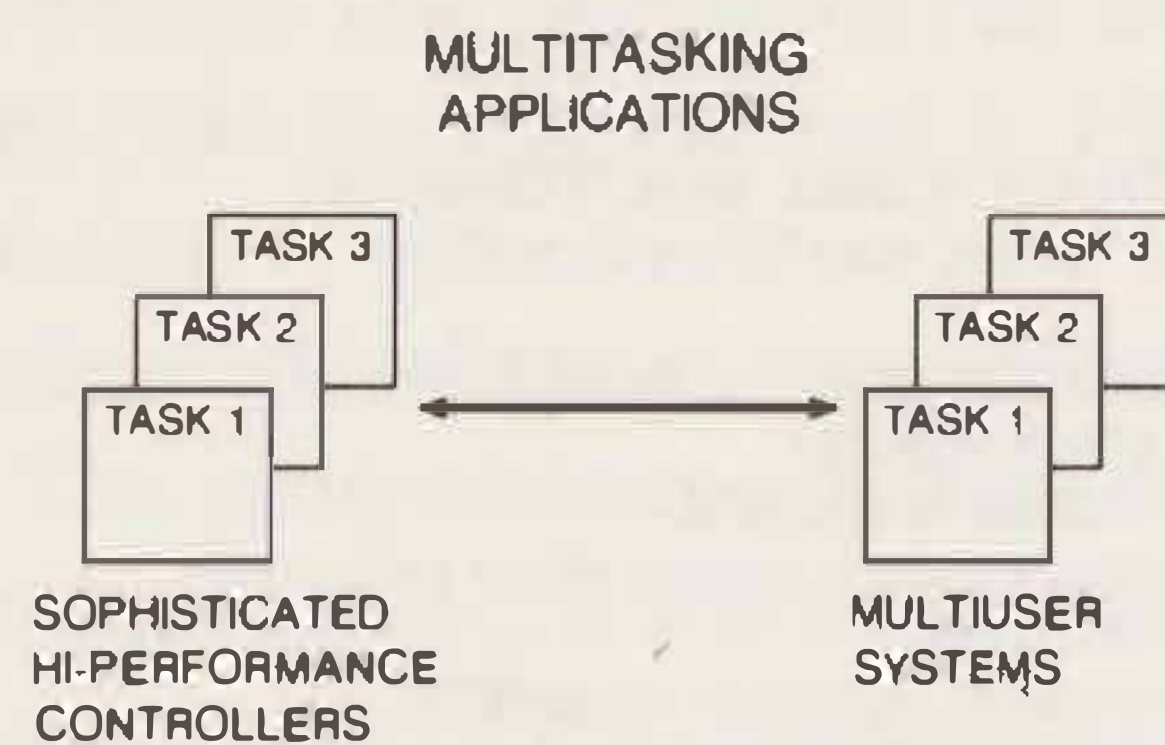
An example of a paged virtual memory system.

Of course, the page size must be reasonable for this situation to be likely. If the pages are too large, the system cannot fit many of them into physical memory. Furthermore, transferring a page to or from disk may take a long time. If pages are too small, the system spends too much time dealing with them. Typical sizes for pages in actual computers are 1K to 4K bytes. The 80386 uses a 4-Kb page. Note that page size is fixed, whereas segment size can vary.

Paging is not always desirable. Applications such as guidance and control that require real-time response often cannot tolerate the delays caused by page swapping. One hardly wants a missile or spacecraft to careen off course because its controller needs extra time occasionally to obtain a new page from disk. Even less critical applications may require that some pages always reside in physical memory. These could include a real-time clock, a high-priority interrupt, and the page fault handler itself. Obviously, the computer is in big trouble if activating the page fault handler causes a page fault.

Virtual memory always adds complexity to a computer system. In the first place, address translation takes extra time or hardware. The same holds for determining whether pages are available and marking usage for swapping algorithms. Furthermore, the operating system must set up the page tables and initialize registers and memory locations. The 80386 provides many features intended to reduce this complexity and

MULTITASKING



WHAT IS MULTITASKING?

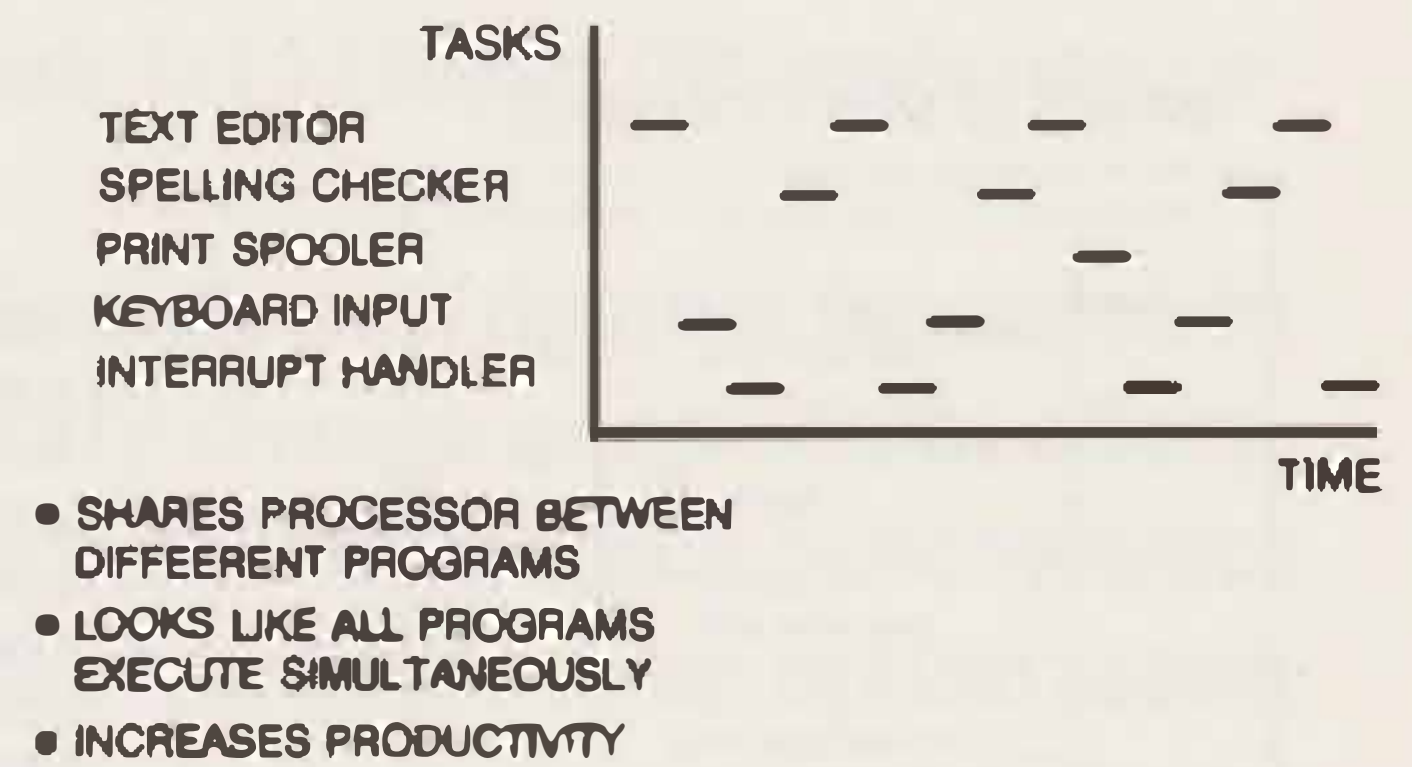


Figure 1-13

Dividing processor time among many tasks.

overhead. We have already noted that it can do address translation in parallel with other activities.

Multitasking

Another key concept in 80386-based systems is multitasking. This refers to running many tasks “at once,” usually by giving each one a slice of CPU time and suspending those that must wait for input/output or other external events. Figure 1-13 gives a crude idea of how multitasking works for a system running five tasks. The system intertwines the execution of tasks, moving from one to another according to priority. In most cases, systems give very high priority to short jobs. Thus the computer can generally finish them quickly without greatly affecting the run times of long jobs.

Each task is an independent entity. It has its own program and data areas, startup procedures, status, and priority. To run tasks efficiently, a computer must be able to:

- Switch rapidly from one task to another. This generally involves saving and loading the entire machine state (registers and other facilities) as shown in Figure 1-14.
- Keep tasks from interfering with one another, while still permitting efficient communications.
- Resolve conflicts and prioritize operations.

The 80386 provides special structures and instructions for holding task status, switching tasks, and creating local and global environments. These hardware facilities greatly reduce the overhead for multitasking.

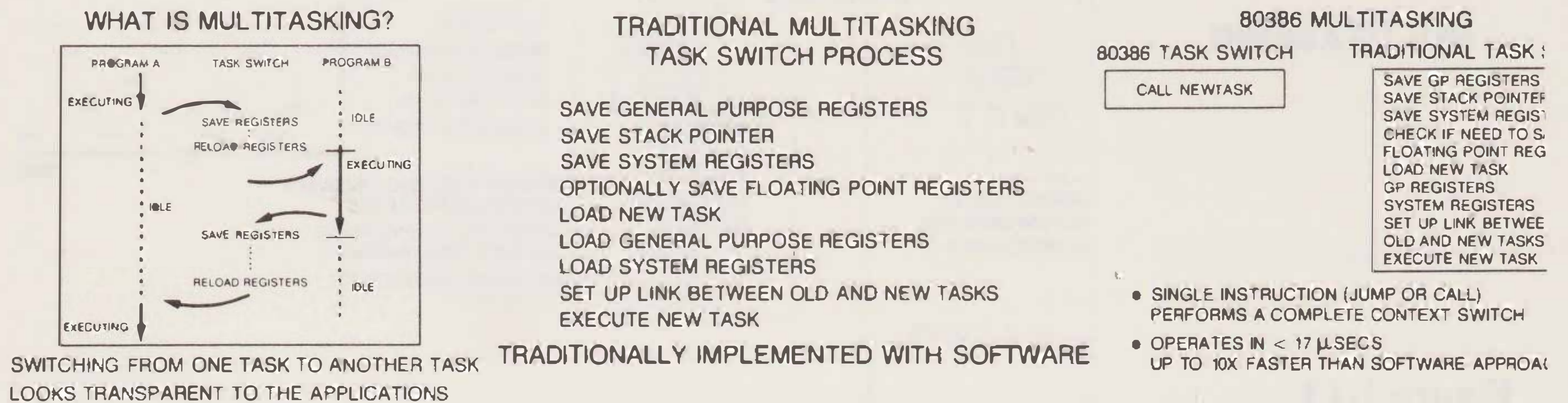


Figure 1-14
A task switch.

Most personal computer users first see the advantage of multitasking when waiting for a long printout. Without multitasking, printing occupies the computer completely. You cannot edit another file or do any other work until the printing is finished. The effect is like waiting at the bank behind someone who has a mere 2,000 transactions to complete. Other tasks that may tie up a computer for a long time include calculating a large spreadsheet, compiling a long program, sorting a database, checking spelling or grammar in a long document, and transferring a large file via a modem.

In multitasking, everyone gets a share of the available processing time (see Figure 1-13). This does not affect tasks such as editing or data input. After all, they would be spending most of their time waiting for I/O anyway. For example, even the fastest typist can enter only about 30 characters per second. The multitasking operating system simply does not run such tasks until their next input is available or their most recent output is finished. Meanwhile, other tasks can use time that would otherwise be wasted. The situation is like having someone do other work while waiting for infrequent service calls. The legendary Maytag repairman can get a lot done this way. And it has little effect on the response time.

Multitasking also is useful in many situations not involving people. For example, a controller for a nuclear power plant may have many jobs to do. It must log the plant's current status, respond to alarms, print reports, provide local displays, and perhaps communicate with a central computer. Multitasking allows it to do all these things at once.

“At once” is figurative here. The processor can actually do only one task at a time. However, short time slots (like the ones shown in Figure 1-11) make the response appear immediate to most users.

Multitasking has a further advantage in many situations. It puts distinct functions in separate, isolated units. The programmer can then readily change one without affecting the others. For example, you might want to change keyboards, add a faster printer or one with color graphics, or replace a communications protocol or an optimization method. If the tasks are truly independent, you should only have to change one of them. This isolation would also apply if some tasks differed, depending on the computer model, operating mode, or optional attachments.

The cost of multitasking is generally quite low. Of course, the operating system uses some processing time for task management. However, in practice, most tasks spend much of their time (perhaps 80 percent or more) waiting for I/O. Multitasking frees this time for productive use. Of course, systems get bogged down if they have too many tasks or if tasks are compute-bound rather than I/O-bound. Overall, multitasking makes all tasks take somewhat longer to execute, but it allows the computer to do much more useful work.

Multitasking does require some arbitration. Two tasks cannot both use the same I/O device or memory area. One must wait for the other to relinquish control. But what happens if each has something the other needs and neither will yield? The operating system must resolve the conflict (called *deadlock*). Another problem is a task that ties up resources and then dies. Perhaps it tried to divide by zero or commit some other unforgivable crime. The operating system must remove the carcass (a nice image!) and free the resources. The operating system may also have to intercept outputs and request inputs to avoid conflicts over I/O devices.

Multiuser Systems

The 80386 also simplifies the implementation of multiuser systems. Although single user systems have become more popular in recent years as computer costs have decreased, multiuser systems are still necessary in some situations. In particular, a multiuser system allows many people to share a common database such as the records for a company, hospital, school, or government agency. Clearly, having everyone keep their own copies of common records would lead to needless confusion and repetition.

Multiuser systems are also essential for online transactions, such as reservation systems, order entry, and banking. Here the system's main purpose is to provide rapid access to common records.

Multiuser systems raise many new problems. Each user must get a slice of time much as in multitasking systems. In a sense, we can regard each user as a task. However, the key element is access to shared data. We need ways to provide this access without allowing the data to be corrupted, read while it is being updated, or accessed improperly.

Protection

Virtual memory, multitasking, and multiuser systems all require protection for key data and operating software. Users cannot be allowed to interfere with processes that affect other people.

We may compare shared computer facilities with a carpool. Fortunately, sharing a computer works better in most cases, although the problems are similar. If you drive to work by yourself, how your car runs, when you come and go, and what route you take are all generally your own business. A flat tire or a dead battery inconveniences only yourself. Furthermore, you can make whatever side trips or extra stops you want.

When you become part of a carpool, however, the situation changes. Now a flat tire inconveniences everyone. Riders do not generally appreciate side trips, extra stops, and late arrivals or departures. Everyone must give up some freedom for the benefit of the group. This is why you don't see many successful carpools in practice.

Shared facilities require an administrator who provides centralized control and recordkeeping. This applies to multiuser computers, networks, shared copying machines or printers, common areas in apartments or condominium complexes, and public parks.

The situation with operating systems is similar. In the single-user single-task environment, you can do almost anything you want. Even overwriting the entire operating system hurts no one but yourself. Furthermore, the operating procedures are usually simple enough so that you can readily get around the system's limitations. With a little experience, of course. If things do not work out, you can always reset the machine or turn it off and start over.

The multitasking or multiuser environment is different. One task or user cannot be allowed to disrupt others or spy on them. Resetting the computer or turning it off is no longer a reasonable solution to a runaway program. It is, however, a great way to meet other users and raise the general level of local hostilities.

What we need is a way to protect programs and data. We must be able to protect operating system software from users, and one user's programs and data from other users. The 80386 provides mechanisms for doing both of these. It thus is well-suited to multiuser and multitasking systems. Furthermore, its hardware protection mechanisms take little or no extra time.

Support for High-Level Languages

Another feature of the 80386 is its support for high-level languages. This is surely essential when we are considering memory capacities in the gigabyte and terabyte range. Presumably, no one will write even a mere gigabyte of assembly or machine code. It would take almost forever to code, debug, and test. It would also require a battalion of programmers to maintain. For that matter, writing a gigabyte of code in a relatively simple high-level language such as BASIC or C would be a formidable job.

How do we make high-level languages run more efficiently? Among the requirements are:

- Ready access to the stack for loading and saving operands. Compilers use the stack because it is ordered and easy to expand.
- A wide variety of indexed and indirect addressing modes. Compilers need these to obtain data through variable pointers.
- Ability to handle many data types. Compilers must be able to deal with bytes, words, double words, bits, bit fields, floating point numbers, and other formats.
- Automatic verification of restrictions such as array bounds, stack limits, and reading and writing limitations.
- Clear separation of program, data, and stack areas.
- A consistent structure that simplifies code generation.

The 80386 provides all these capabilities and features in a convenient package.

COMPARISONS WITH PREVIOUS PROCESSORS

The 80386 processor is fully compatible with the 8086, 8088, and 80286 processors. The major advances, as mentioned earlier, are:

- 32-bit facilities and data paths throughout
- Greatly expanded memory capacity

- Added hardware to allow more parallel operations through greater pipelining
- Special hardware to speed up shifting, multiplication and division, and address calculations

Overall, we may estimate the 80386's performance at 16 MHz as follows:

- Three times that of an 80286 running at 8 MHz
- Ten times that of an 8086 or 8088 running at 6 MHz

This is a rough guide only, and your actual mileage may vary. Seriously, the multiplying factor depends on the application, the instruction mix, and the speed of the computer's other components (such as memory and I/O). For example, early 80386-based CAE workstations draw complex screens, connect components, and transfer schematics from disk to screen 2 to 4 times as fast as their 80286-based predecessors.

As mentioned previously, the 80386 has 256 times the physical memory capacity of the 80286 and 4096 times that of the 8086 and 8088. Its virtual memory capacity is 64K times that of the 80286 (the 8086 and 8088 do not support virtual memory). The result is an awesome 64-terabyte (TB) capacity. This would require a mere 130,000,000 2-Mb boards! Or, since the 64 Tb is virtual, you could use 3 million 100-Mb disks. Despite what magazine advertisements may claim, you probably shouldn't send in your check for the first terabyte PC right away. The 80386 also allows segments as large as 4 Gb, as compared to the 64K maximum in earlier processors.

Of course, many of these new features apply only to programs written specifically for an 80386. MS-DOS programs, for example, are limited to 64K segments and 640K of memory, even on an 80386-based computer. After all, they can only use features that are compatible with the 8088 and 8086 processors. Similarly, OS/2 programs are limited to 64K segments and 16 Mb of memory. They can only use features that are compatible with the 80286. The advantages of the 80386 in running old programs are its higher clock speed, more extensive pipelining, 32-bit data paths and facilities, and faster mathematical hardware.

WHAT'S NEXT IN MICROPROCESSORS?

Obviously, the 80386 is not the last word from Intel or other vendors. There are many demands for large amounts of low-cost processing power that even it cannot satisfy. Fortunately, there are well-known techniques microprocessor designers can use to increase throughput. Among the ones you may see in future devices are:

- Larger on-chip instruction queues. These would allow even more pipelining than is now possible.
- On-chip memories (called *caches*) for extremely high-speed access. Program caches could hold an entire loop or sets of instructions from all recent branch addresses. Either approach would avoid the need to refill the pipeline after every jump.
- On-board stack caches to hold the top locations of the stack. This would speed up subroutines that keep most of their parameters and variable data on the stack.
- Separate data and instruction memories. This would allow overlapping of data transfers with instruction fetches.
- Greater optimization of instructions and instruction sets.
- More registers to allow more on-chip storage.

Of course, we will also see higher clock speeds, pipelines with more stages, and faster memory and I/O chips.

A more general approach is to design a processor with fewer but faster-executing instructions. It would need less decoding circuitry and could have more registers and arithmetic circuits. We refer to such a processor as a reduced-instruction set (RISC) machine.

SUMMARY

The 80386 microprocessor represents a significant advance in low-cost computing power. It can run the large backlog of 8086 (MS-DOS) software much faster than can the 8088, 8086, and 80286 microprocessors. It also provides access to a huge amount of memory (4 Gb) but only in its native operating mode.

The 80386's native architecture simplifies the implementation of many key features of advanced computer systems, including:

- Virtual memory
- Multitasking
- Multiuser systems
- High-level languages and operating systems
- Security, protection, and privacy

Among its primary application areas are personal computers, CAD/CAM/CAE systems, robotics, artificial intelligence, and signal processing.

80386 Architecture and Instruction Set

*In my experience, if you have to keep
the lavatory door shut by extending your
left leg, it's modern architecture.*

Nancy Banks-Smith, *Guardian*, 20 February 1979

*I've finally learned what "upward compatible" means.
It means we get to keep all our old mistakes.*

Dennis Van Tassel

This chapter describes the 80386's basic architecture. It starts with the registers and then covers data types, addressing modes, and instructions. It deals with frequently used addressing modes and instructions first. The last sections discuss 8086 and 80286 compatibility and introduce common assembler directives.

The chapter presents features used by applications programmers as opposed to those used only by systems programmers. Of course, as students of the data processing cul-

ture know, “Real programmers don’t write applications programs. . . . Applications programs are for dullards who can’t do systems programming.”

NOTATION

This book describes 80386 facilities and programs using notation from Microsoft’s Macro Assembler. The key elements after a number are:

B Binary

D (or no designation) Decimal

H Hexadecimal. Hexadecimal numbers that begin with a letter digit (A through F) require an initial zero to distinguish them from symbolic names. For example, we would write FF as 0FFH.

The default case (that is, unmarked) is decimal. Other symbols are:

: After a label associated with an instruction statement or between segment register designations or segment numbers and offsets

; Before a comment

‘ Around characters (before and after a string)

[] Around a memory address

Names, numbers, and expressions not enclosed in brackets are taken to be data values.

REGISTERS

Figures 2-1 and 2-2 show the 80386’s basic architecture. Figure 2-2 omits the segment registers that simply position program, data, and stack areas in memory. The general-purpose or user registers (see Figure 2-2) are:

- EAX, the primary accumulator. It is also the holding place for most data being moved into and out of the processor.
- EBX, the base (address) register. It often holds addresses for indexing and indirection.
- ECX, the count register. Its main purpose is to hold the number of iterations or shifts.
- EDX, the data register. It serves as an extension of the accumulator and holds port addresses for input and output.

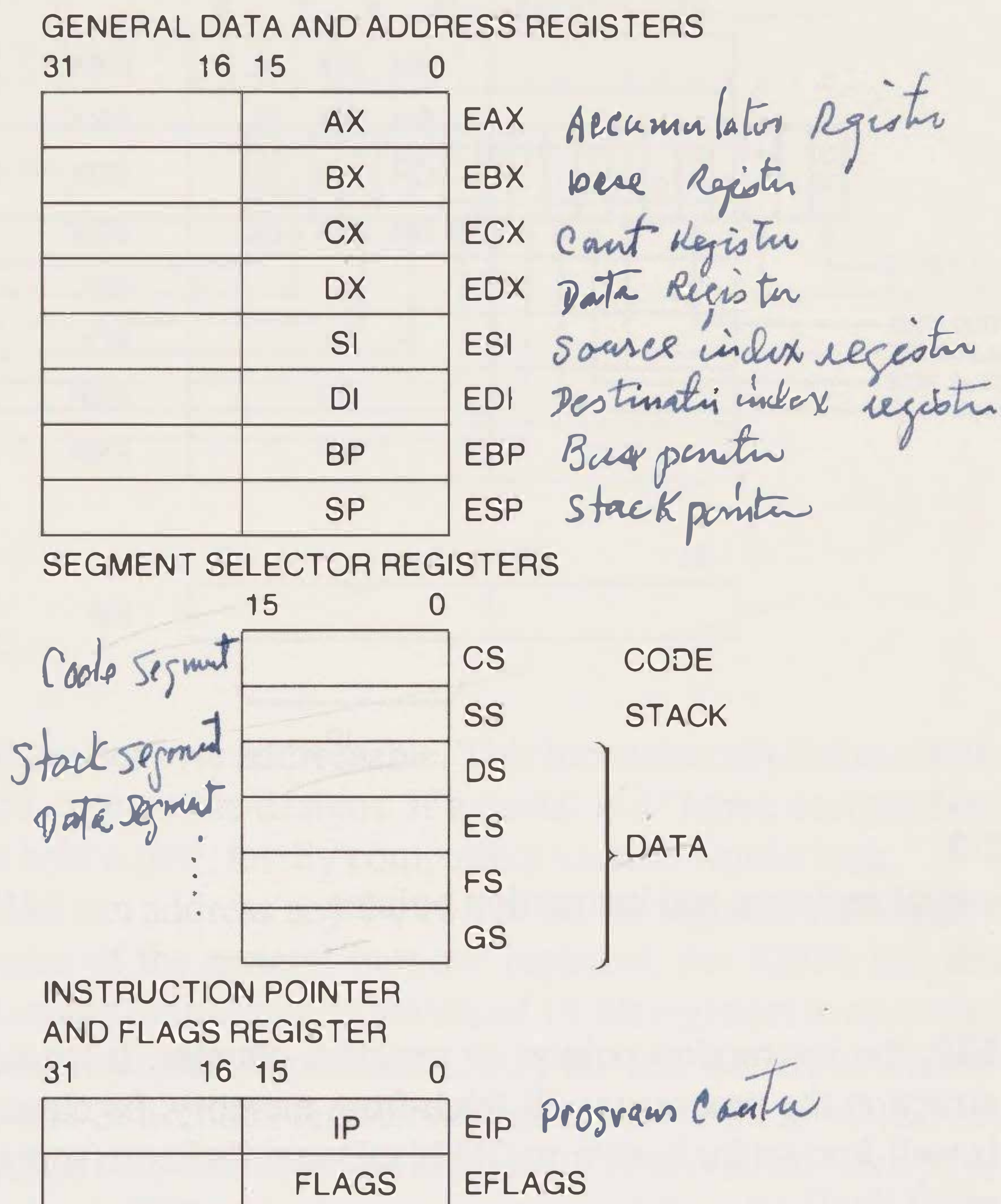


Figure 2-1

80386 base architecture registers.

- ESI and EDI, the source and destination index registers. They often hold pointers for array and string manipulation. The names come from their roles in string instructions.
- EBP, the base pointer. Its main use is in accessing data on the stack.
- ESP, the stack pointer (nothing out of this world, unfortunately). Its main use is in moving data and addresses to and from the stack.

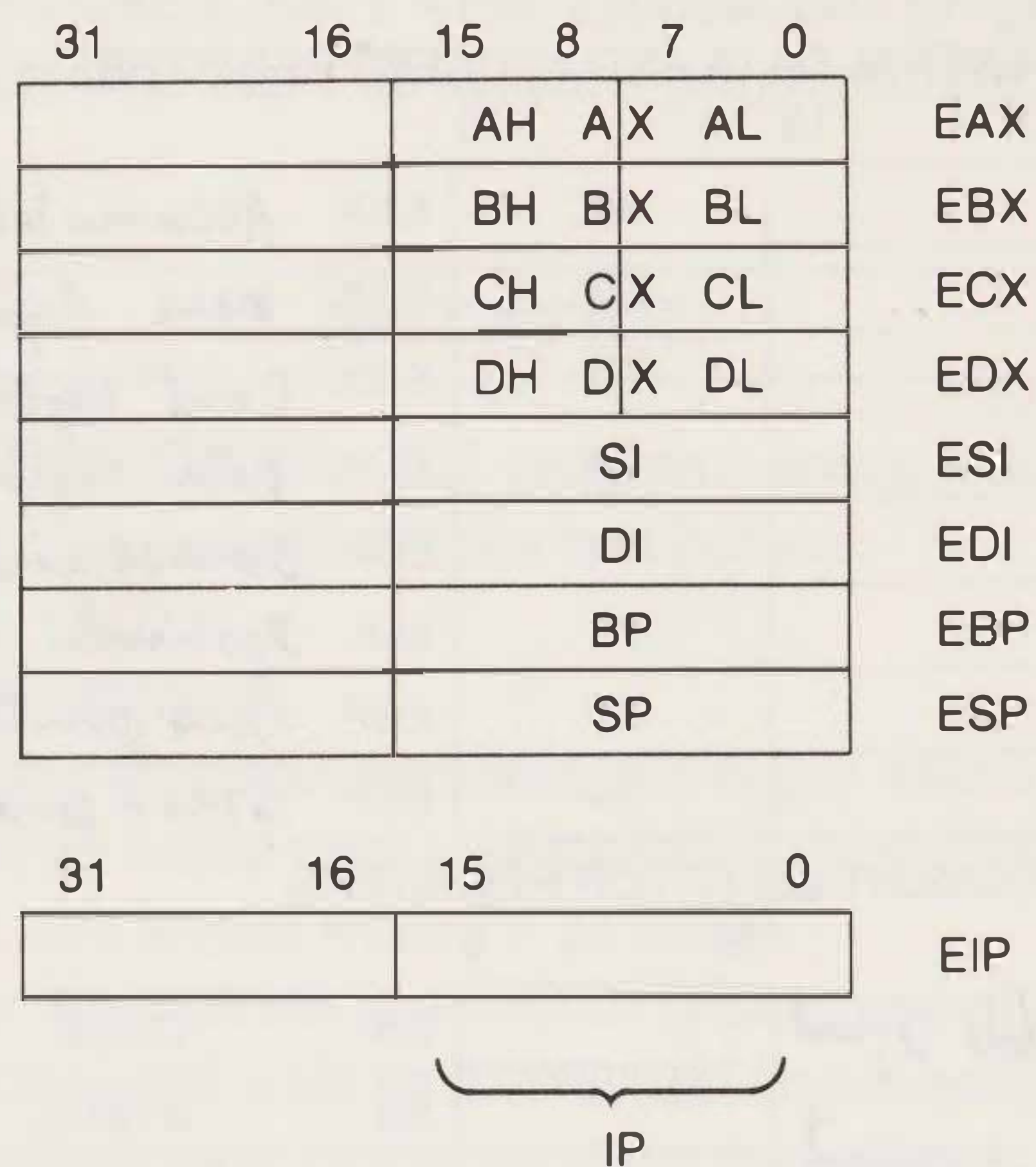


Figure 2-2
80386 general registers and instruction pointer.

- **EIP**, the instruction pointer or program counter. It locates the next instruction the processor will fetch from memory. Its close relative, **EIO**, is well-known for its role on Old MacDonald's Farm (sorry, but I couldn't resist this!).

The initial **E** in the names indicates an "extended" or 32-bit register. As Figure 2-1 shows, the *low words* (bits 0 through 15) of these registers have the same names without the **E**. The 16-bit registers, which can be accessed separately, are carryovers from earlier 16-bit processors (8088, 8086, and 80286). Note that all general-purpose registers have 16-bit subunits.

As Figure 2-2 shows, parts of some registers are byte addressable. As this feature is also derived from earlier processors, it applies only to the low words. The byte-length registers are:

- **AH** and **AL** (bits 8 through 15 and 0 through 7 of **EAX**)
- **BH** and **BL** (bits 8 through 15 and 0 through 7 of **EBX**)
- **CH** and **CL** (bits 8 through 15 and 0 through 7 of **ECX**)
- **DH** and **DL** (bits 8 through 15 and 0 through 7 of **EDX**)

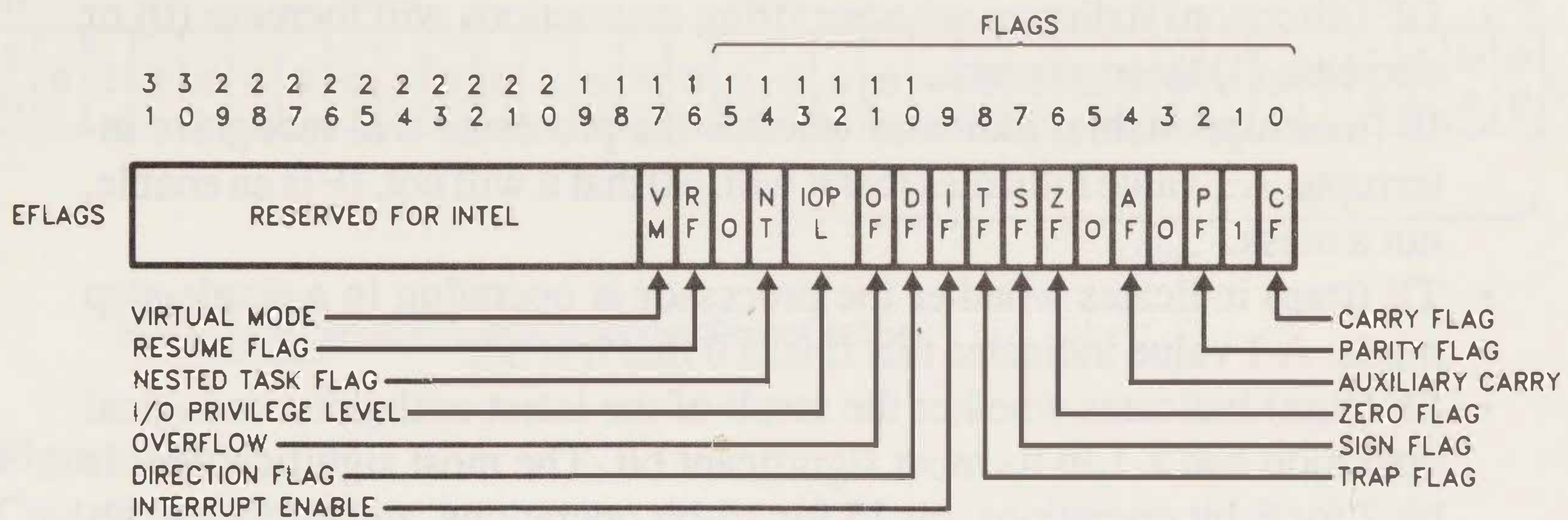


Figure 2-3
80386 flags register.

SI, DI, BP, SP, and IP are not byte addressable. This inconsistency is the result of three generations of upward-compatible designs. If a camel is a “horse designed by a committee,” just imagine how a new, totally compatible version would look.

In general, the 80386 can address any 8-, 16-, or 32-bit register. Although we have mentioned primary uses of the general-purpose registers, the 80386 has few actual restrictions. There are some limitations on the use of 16-bit registers in accordance with the 8086 and 80286 architectures. In particular, only BP and BX are available as base registers and only DI and SI as index registers. Even in the 32-bit mode, however, opposing historical tradition may cost time and memory. Conservatism has a payoff here.

The instruction pointer (EIP) or program counter (PC) differs from other user registers. This is why it appears in a separate part of Figure 2-1. The difference is that the processor controls EIP most of the time. In particular, the processor increases it automatically after fetching an instruction from memory. Like most processors, the 80386 thus executes instructions sequentially unless specifically told to do otherwise. The programmer can control EIP through instructions (calls, jumps, returns, and software interrupts or traps) that change its value explicitly.

The bottom part of Figure 2-1 also shows a status register called EFLAGS. The low word of this register, called FLAGS, is compatible with earlier processors. Figure 2-3 shows EFLAGS in detail. For now, we will be concerned only with the following bits:

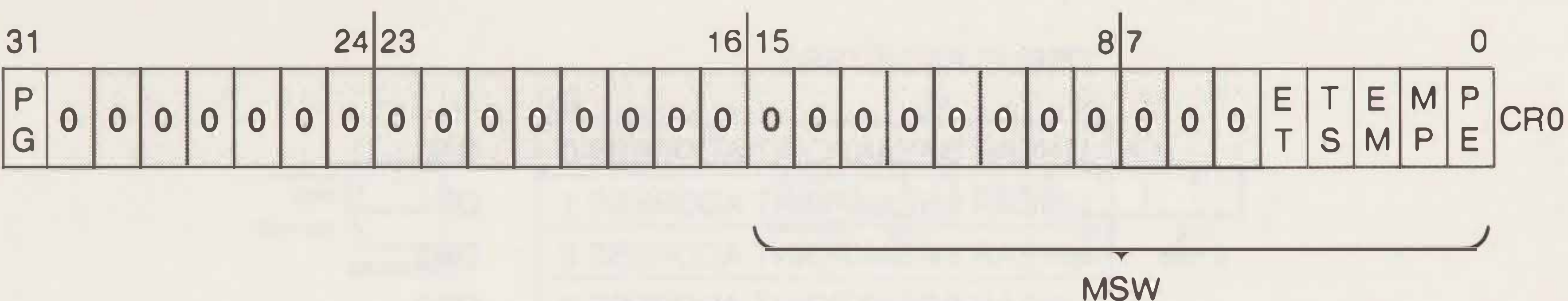
- OF (overflow) indicates whether the latest arithmetic operation or shift produced an (arithmetic) overflow.

- DF (direction) indicates whether string instructions will increase (0) or decrease (1) their pointers.
- IF (interrupt enable) indicates whether the processor will recognize interrupts. A 1 value indicates that it will, a 0 that it will not. IF is an enable, not a mask.
- TF (trap) indicates whether the processor is operating in a single-step mode. A 1 value indicates that it is, a 0 that it is not.
- SF (sign) indicates whether the result of the latest arithmetic or logical operation had a 1 in its most significant bit. The most significant bit is bit 7 for 8-bit operations, bit 15 for 16-bit operations, and bit 31 for 32-bit operations.
- ZF (zero) indicates whether the result of the latest arithmetic or logical operation was 0. Note that ZF is 1 if the result was zero, and 0 if it was not. Although this flag is standard in processor architecture, it remains a source of confusion.
- AF (auxiliary carry) indicates whether the latest arithmetic operation produced a carry from bit 3. AF's main use is in decimal arithmetic.
- PF (even parity) indicates whether the result of the latest arithmetic or logical operation had even parity. PF is 1 if it did and 0 if it did not. Even parity means that the number of 1 bits is even. PF reflects only the low byte (bits 0 through 7) of the result, regardless of how many bits the operation involves.
- CF (carry) indicates whether the latest arithmetic, logical, or shift instruction produced a carry. Logical instructions such as AND and OR cannot produce a carry, so they always clear this flag. Bit manipulation instructions use CF to store the tested bit's value.

The common flags are (from right to left): Carry, Zero, and Sign (CF, ZF, and SF). Programs often use their values to choose between alternative paths. Carry also transfers a bit between operations in multiple-precision arithmetic.

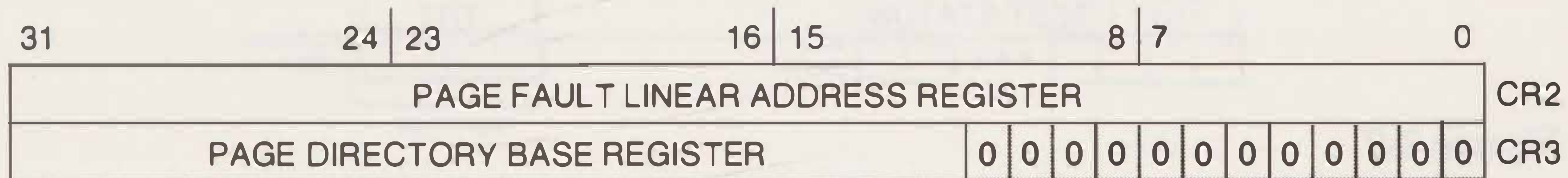
The middle part of Figure 2-1 shows the segment registers. They select subdivisions of memory called *segments*. Note that they are 16-bit registers, even though the 80386 is a 32-bit processor. The segment registers are:

- CS, the code segment register, selects the segment from which the processor will obtain instructions.
- SS, the stack segment register, selects the segment occupied by the stack.
- DS, ES, FS, and GS, the data segment registers, select the current segments used for data. We call DS the data segment register and ES the



NOTE: 0 indicates Intel reserved: Do not define.

Figure 2-4 Control register 0.



NOTE: 0 indicates Intel reserved: Do not define

Figure 2-5
Control registers 2 and 3.

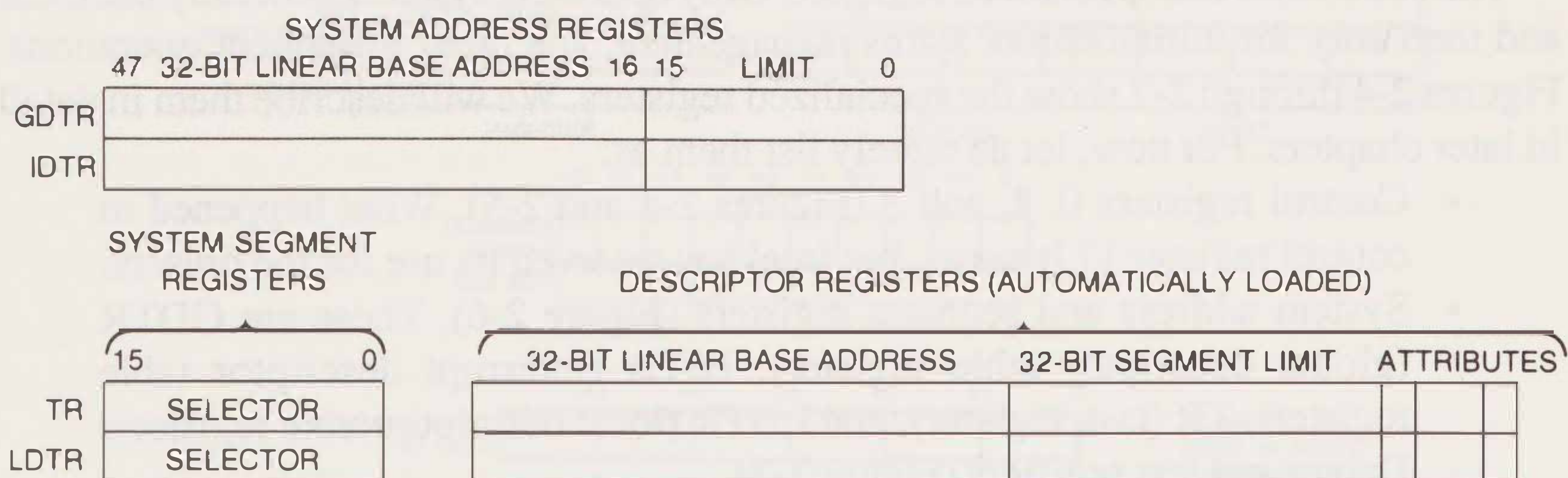


Figure 2-6
System address and system segment registers.

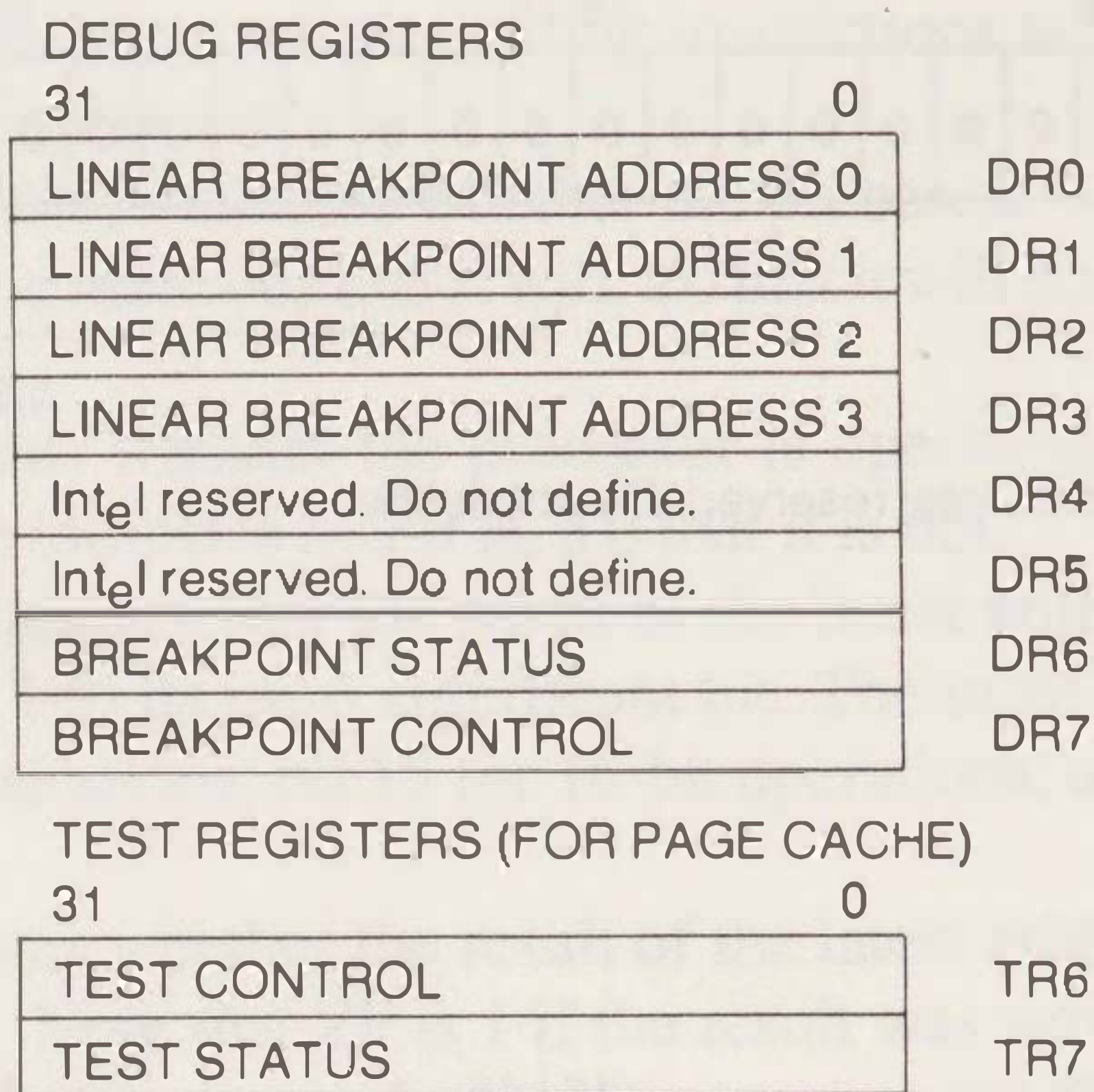


Figure 2-7
Debug and test registers.

extra (data) segment register. They were the only data segment registers in earlier processors. The names FS and GS, the additions to the 80386, have no significance other than following ES alphabetically. Fortunately, Intel did not go backward and give us AS and BS.

The 80386 also has specialized registers. Only operating systems generally use them and then only for initialization, status management, and other infrequent operations. Figures 2-4 through 2-7 show the specialized registers. We will describe them in detail in later chapters. For now, let us merely list them as:

- Control registers 0, 2, and 3 (Figures 2-4 and 2-5). What happened to control register 1? It exists, but Intel has reserved its use for the present.
- System address and segment registers (Figure 2-6). These are GDTR (global descriptor table register), IDTR (interrupt descriptor table register), TR (task register), and LDTR (local descriptor table register).
- Debug and test registers (Figure 2-7).

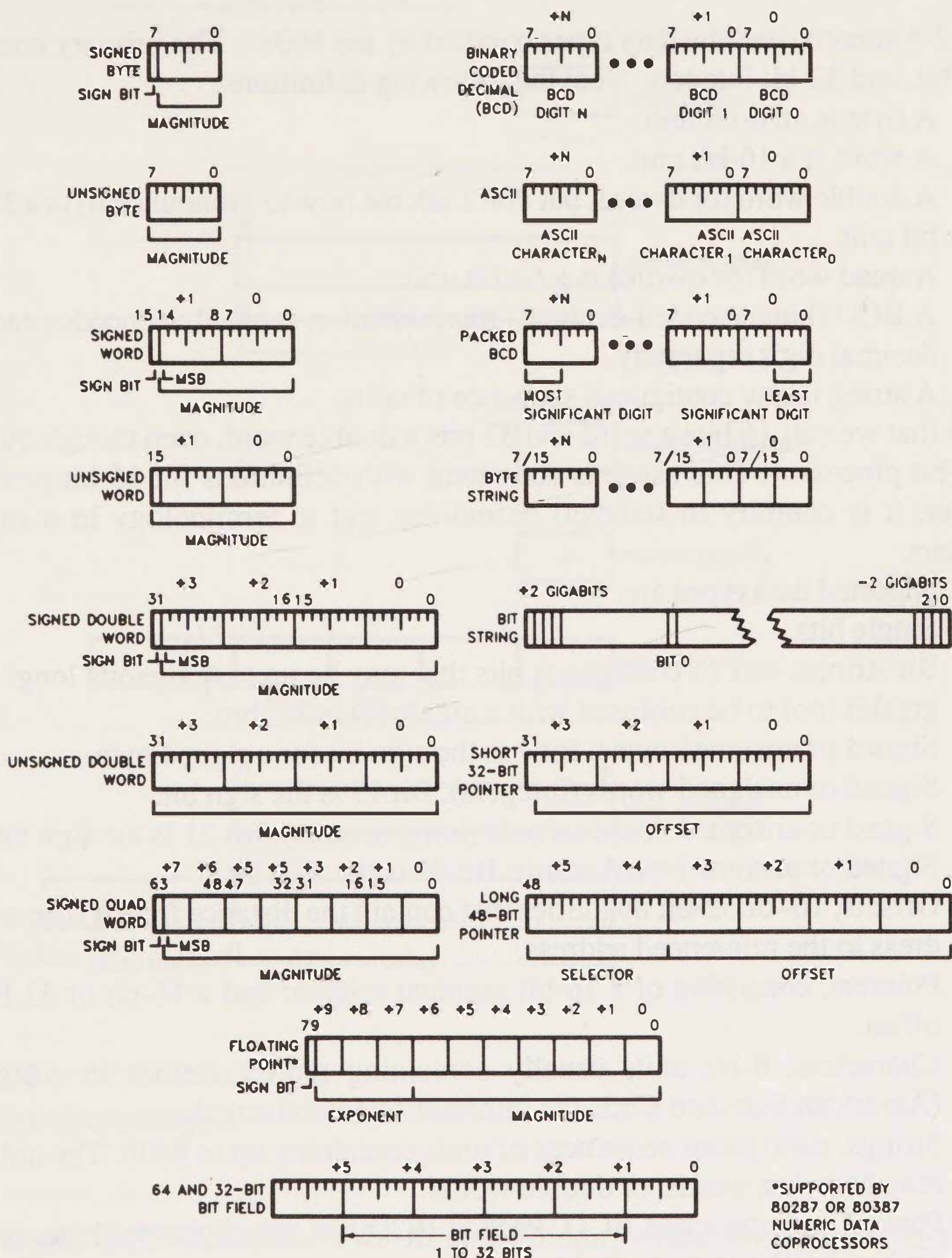


Figure 2-8
80386 supported data types.

DATA TYPES

Figure 2-8 summarizes the data types handled by the 80386. The primary ones are 8-bit, 16-bit, and 32-bit integers. Note the following definitions:

- A byte is an 8-bit unit.
- A word is a 16-bit unit.
- A double word (or dword, but don't ask me how to pronounce it) is a 32-bit unit.
- A quad word (or qword) is a 64-bit unit.
- A BCD (binary-coded-decimal) representation is one that encodes each decimal digit separately.
- A string is any contiguous sequence of units.

Note that we call 16 bits a word and 32 bits a double word, even though the 80386 is a 32-bit processor. This usage is consistent with definitions for 16-bit processors. However, it is contrary to standard definitions and to terminology in many other computers.

The supported data types are:

- Single bits.
- Bit strings, sets of contiguous bits that may be up to 4 gigabits long. A gigabit (not to be confused with a gigabyte) is 2^{30} bits.
- Signed or unsigned bytes. Bit 7 is the sign bit for a signed byte.
- Signed or unsigned words (integers). Bit 15 is the sign bit.
- Signed or unsigned double words (long integers). Bit 31 is the sign bit.
- Signed or unsigned quad words. Bit 63 is the sign bit.
- Offsets, 16- or 32-bit quantities that contain the distance from a base address to the referenced address.
- Pointers, consisting of a 16-bit segment selector and a 16-bit or 32-bit offset.
- Characters, 8-bit units usually containing representations in ASCII (American Standard Code for Information Interchange).
- Strings, contiguous sequences of units containing up to 4 Gb. The units may be bytes, words, or double words.
- Packed and unpacked BCD. Packed BCD has two digits per byte, unpacked BCD one digit per byte.
- Floating point. An 80287 or 80387 numerical coprocessor (see Chapter 8 and Appendixes G, H, and I) handles 32-bit, 64-bit, and 80-bit representations of real numbers. The formats come from the IEEE 754

80386 Architecture and Instruction Set

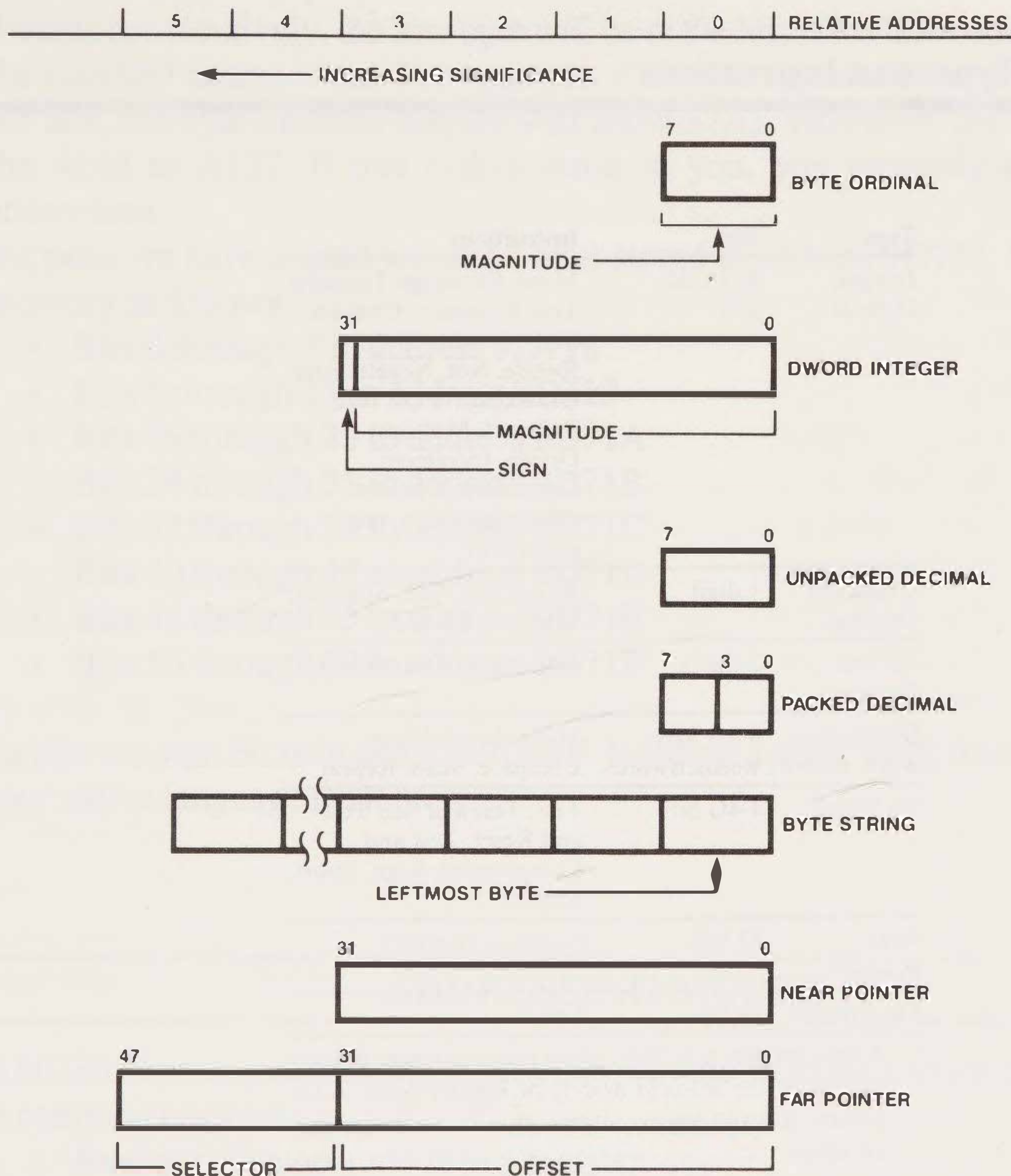


Figure 2-9
80386 data type storage.

standard (*IEEE Standard for Binary Floating-Point Arithmetic*, IEEE, New York, 1985).

As mentioned earlier, the primary data types are bytes, words, and double words. Most instructions apply to them as shown in Table 2-1. Only a few instructions operate on the other types.

Table 1
Principal Data Types and Instructions

Type	Size	Instructions
Integer, Ordinal	8, 16, 32 bits	Move, Exchange, Translate, Test, Compare, Convert, Shift, Double Shift, Rotate, Not, Negate, And, Or, Exclusive Or, Add, Subtract, Multiply, Divide, Increment, Decrement, Convert (Move with sign/zero extension)
Unpacked Decimal	1 digit	Adjust for: Add, Subtract, Multiply, Divide
Packed Decimal	2 digits	Adjust for: Add, Subtract
String (byte, word, dword)	0-4G bytes, words, dwords	Move, Load, Store, Compare, Scan, Repeat
Bit String	1-4G bits	Test, Test and Set, Test and Reset, Test and Complement, Scan, Insert, Extract
Near Pointer ¹	32 bits	(Same as Ordinal)
Far Pointer	48 bits	Load

1. A near pointer is a 32-bit offset into a segment defined by one of the segment/descriptor register pairs. A far pointer is a full logical address, that is, a selector and an offset.

Figure 2-9 shows how data types appear in memory. In general, the low byte is always at the lowest (starting or base) address. Bytes of increasing significance are at successively higher addresses. Although this may seem backward at first, it does allow arithmetic, logical, array, and string operations to begin with the low byte at the lowest address.

Here are examples of 80386 data storage:

1. Suppose we have a word stored at address C0000 hex. Its low byte is in address C0000, and its high byte is in address C0001. If a dump utility displays these ad-

addresses consecutively, the low byte will be at the left. This is, of course, opposite to the standard arrangement. For example, if C0000 contains 37 and C0001 contains A1 hex, the byte-oriented display will show 37A1. However, the 80386 will read the word as A137. If this makes sense to you, you probably should be a tax accountant.

2. Suppose we have a quad word (64 bits) stored at address 9D718. It will appear in memory as follows:

- Bits 0 through 7 in address 9D718
- Bits 8 through 15 in address 9D719
- Bits 16 through 23 in address 9D71A
- Bits 24 through 31 in address 9D71B
- Bits 32 through 39 in address 9D71C
- Bits 40 through 47 in address 9D71D
- Bits 48 through 55 in address 9D71E
- Bits 56 through 63 in address 9D71F

Note that the sign bit is in address 9D71F. Reading a quad word from a dump is easy if your native language is Hebrew.

ADDRESSING MODES

The 80386 has many addressing modes that provide different ways to access operands. The common ones are:

- Register. The operand is in a register.
- Immediate. The operand is part of the instruction, usually immediately following the operation code.
- Direct. The operand's address is part of the instruction.
- Register indirect. The operand's address is in a base or index register.
- Index. The operand's address is the sum of an index register and a displacement. The displacement is part of the instruction.
- Based index with displacement. The operand's address is the sum of an index register, a base register, and a displacement.

Note the following definitions:

A *displacement* is an 8-bit, 16-bit, or 32-bit immediate value following an operation code. This is a fixed part of program memory.

A *base* is the contents of any general-purpose register. The registers most often used for this purpose are EBX and EBP.

An *index* is the contents of any general-purpose register except ESP. The registers most often used for this purpose are ESI and EDI.

Let us now look at examples of the common addressing modes:

1. `MOV EBX,EAX`. This instruction uses register addressing to move the contents of register EAX to register EBX. We will explain the seemingly backward order of the operands shortly.
2. `ADD AL,5`. This instruction uses immediate addressing to add the number 5 to register AL. Note that an unmarked number is treated as data, not as an address. The same holds for an unmarked name or expression in standard assembler notation.
3. `OR AL,[2000H]`. This instruction uses direct addressing to logically OR the contents of memory location 2000 (hex) with register AL. Note that brackets around a number or expression indicate that it is an address, not data.
4. `MOV ECX,[EBX]`. This instruction uses register indirect addressing to move data from the address in EBX to register ECX. If, for example, EBX contains 1000 (hex), ECX ends up with the contents of memory locations 1000 (low byte) through 1003 (high byte).
5. `MOV AL,100H[ESI]`. This instruction uses the based index mode to load AL from the address obtained by adding 100 (hex) to the contents of ESI. If, for example, $[ESI] = 1700$ hex, the effective address is $1700 + 100 = 1800$ hex. An alternative (perhaps clearer) notation is `MOV AL,[ESI+100H]`.
6. `MOV EAX,5[EBX+ESI]`. This instruction uses the based index mode with a displacement. The effective address is the sum of EBX, ESI, and 5 (the displacement). If, for example, $[EBX] = 1D00$ (hex) and $[ESI] = 0200$ (hex), the effective address is $1D00 + 0200 + 5 = 1F05$ (hex). The new contents of EAX come from that address and the next three higher addresses (that is, 1F05 through 1F08). Alternative notations are `MOV EAX,[EBX+ESI+5]` and `MOV EAX,5[EBX][ESI]`.

Less common addressing modes use a scaled index. Here the processor multiplies the index by a scaling factor (2, 4, or 8) and uses the product to compute the effective address. This mode is useful for accessing arrays with multibyte entries. For example, suppose we have an array of 32-bit addresses. If its base address is in EBX and the element number is in ESI, the following instruction will load an element into EAX:

`MOV EAX,[EBX+ESI*4]`

Table 2-2
80386 Addressing Modes

Addressing Mode	Form	Effective Address
Register Immediate	r1,r2 data	Register Address immediately after operation code contains data
Direct Register Indirect Based	addr [base] disp[base]	Addr Contents of base register Contents of base register + disp
Index	disp[index]	Contents of index register + disp
Scaled Index	disp[index*scale]	Scale factor \times Contents of index register + disp
Based Index	[base+index]	Contents of base register + Contents of index register
Based Scaled Index	[base+index*scale]	Scale factor \times Contents of index register + Contents of base register
Based Index with Displacement	disp[base+index]	Contents of base register + Contents of index register + disp
Based Scaled Index with Displacement	disp[base+index *scale]	Contents of base register + Scale factor \times Contents of index register + disp

Scaled indexing is necessary because each element occupies 4 bytes of memory. In previous processors, a multiplication or shift was necessary to convert the element number into an actual offset from the base address.

Table 2-2 summarizes the 80386's addressing modes. Note that any 32-bit general-purpose register can serve as a base or index register, except that ESP cannot be an index register. The base and index registers may be the same. The scale factor can only be 2, 4, or 8.

One result of the 80386's increased pipelining is that it can calculate effective addresses while doing other operations. Most addressing modes therefore require no extra clock cycles. The only exceptions are the based index modes with displacement. They take one extra clock cycle.

INSTRUCTION SET

Table 2-3 contains a complete list of the 80386's instructions. Before describing the set in detail, let us first consider the most frequently used instructions. A rule of thumb is that 20 percent of an instruction set (or a language's set of statements) makes up 80 percent of most programs. In assembly language, the rule is probably even stricter. Generally speaking, 10 percent of most processors' instruction sets makes up 90 percent of assembly language programs. This is not to say that the other instructions are unnecessary but just that they are uncommon. This observation is one reason behind the development of RISC (reduced-instruction-set) computers.

Table 2-4 lists the 80386's frequently used instructions. We may characterize them further as follows:

- Data Transfer
 - IN input
 - LEA load effective address
 - MOV move
 - OUT output
 - POP load from stack
 - PUSH store on stack
- Arithmetic/Logical
 - ADC add with carry
 - ADD add
 - AND logical AND
 - CMP compare

Table 2-3
Complete 80386 Instruction Set

Instruction	Meaning	Assembler Format	
AAA	ASCII Adjust after Add	AAA	
AAD	ASCII Adjust before Divide	AAD	
AAM	ASCII Adjust after Multiply	AAM	
AAS	ASCII Adjust after Subtract	AAS	
ADC	Add with Carry	ADC	dest,src
ADD	Add	ADD	dest,src
AND	Logical AND	AND	dest,src
ARPL	Adjust RPL Field of Selector	ARPL	sel,reg
BOUND	Check Array Bounds	BOUND	reg,bound
BSF	Bit Scan Forward	BSF	dest,src
BSR	Bit Scan Reverse	BSR	dest,src
BT	Bit Test	BT	base,offset
BTC	Bit Test and Complement	BTC	base,offset
BTR	Bit Test and Reset	BTR	base,offset
BTS	Bit Test and Set	BTS	base,offset
CALL	Call Procedure	CALL	dest
CBW	Convert Byte to Word	CBW	
CDQ	Convert Double Word to Quad Word	CDQ	
CLC	Clear Carry Flag	CLC	
CLD	Clear Direction Flag	CLD	
CLI	Clear Interrupt Flag	CLI	
CLTS	Clear Task-Switched Flag	CLTS	
CMC	Complement Carry Flag	CMC	
CMP	Compare	CMP	dest,src
CMPS	Compare Strings	CMPS	dest,src
CWD	Convert Word to Double Word*	CWD	
CWDE	Convert Word to Double Word*	CWDE	
DAA	Decimal Adjust after Add	DAA	
DAS	Decimal Adjust after Subtract	DAS	
DEC	Decrement by 1	DEC	dest
DIV	Unsigned Divide	DIV	acc,src
ENTER	Make Stack Frame for Procedure	ENTER	storage,level

Table 2-3 (continued)
Complete 80386 Instruction Set

Instruction	Meaning	Assembler Format	
ESC	Escape	ESC	
HLT	Halt	HLT	
IDIV	Signed Divide	IDIV	acc,src
IMUL	Signed Multiply	IMUL	acc,src
IN	Input from Port	IN	acc,port
INC	Increment by 1	INC	dest
INT	Software Interrupt (Trap)	INT	inttype
INTO	Interrupt If Overflow	INTO	
IRET	Interrupt Return	IRET	
JA	Jump If Above	JA	dest
JAE	Jump If Above or Equal	JAE	dest
JB	Jump If Below	JB	dest
JBE	Jump If Below or Equal	JBE	dest
JC	Jump If Carry	JC	dest
JCXZ	Jump If CX Is Zero	JCXZ	dest
JE	Jump If Equal	JE	dest
JECXZ	Jump If ECX Is Zero	JECXZ	dest
JG	Jump If Greater	JG	dest
JGE	Jump If Greater or Equal	JGE	dest
JL	Jump If Less	JL	dest
JLE	Jump If Less or Equal	JLE	dest
JMP	Jump Unconditionally	JMP	dest
JNA	Jump If Not Above	JNA	dest
JNAE	Jump If Not Above or Equal	JNAE	dest
JNB	Jump If Not Below	JNB	dest
JNBE	Jump If Not Below or Equal	JNBE	dest
JNC	Jump If No Carry	JNC	dest
JNE	Jump If Not Equal	JNE	dest
JNG	Jump If Not Greater	JNG	dest
JNGE	Jump If Not Greater or Equal	JNGE	dest
JNL	Jump If Not Less	JNL	dest
JNLE	Jump If Not Less or Equal	JNLE	dest

Table 2-3 (continued)
Complete 80386 Instruction Set

Instruction	Meaning	Assembler Format	
JNO	Jump If No Overflow	JNO	dest
JNP	Jump If Parity Odd	JNP	dest
JNS	Jump If Sign Positive	JNS	dest
JNZ	Jump If Not Zero	JNZ	dest
JO	Jump If Overflow	JO	dest
JP	Jump If Parity Even	JP	dest
JPE	Jump If Parity Even	JPE	dest
JPO	Jump If Parity Odd	JPO	dest
JS	Jump If Sign Negative	JS	dest
JZ	Jump If Zero	JZ	dest
LAHF	Load Flags into AH Register	LAHF	
LAR	Load Access Rights Byte	LAR	reg,src
LDS	Load DS Register	LDS	reg,src
LEA	Load Effective Address	LEA	reg,src
LEAVE	Leave Procedure	LEAVE	
LES	Load ES Register	LES	reg,src
LFS	Load FS Register	LFS	reg,src
LGS	Load GS Register	LGS	reg,src
LGDT	Load GDT Register	LGDT	src
LIDT	Load IDT Register	LIDT	src
LLDT	Load LDT Register	LLDT	src
LMSW	Load Machine Status Word	LMSW	src
LOCK	Lock Bus	LOCK	
LODS	Load String	LODS	src
LOOP	Loop with CX Counter	LOOP	dest
LOOPE	Loop If Equal	LOOPE	dest
LOOPNE	Loop If Not Equal	LOOPNE	dest
LOOPNZ	Loop If Not Zero	LOOPNZ	dest
LOOPZ	Loop If Zero	LOOPZ	dest
LSL	Load Segment Limit	LSL	reg,src
LSS	Load SS Register	LSS	reg,src
LTR	Load Task Register	LTR	src

Table 2-3 (continued)
Complete 80386 Instruction Set

Instruction	Meaning	Assembler Format
MOV	Move Data	MOV dest,src
MOV	Move to/from Special Regs	MOV dest,src
MOVS	Move String	MOVS dest,src
MOVSX	Move with Sign-Extend	MOVSX reg,src
MOVZX	Move with Zero-Extend	MOVZX reg,src
MUL	Unsigned Multiply	MUL acc,src
NEG	2's Complement Negation	NEG dest
NOP	No Operation	NOP
NOT	1's Complement Negation	NOT dest
OR	Logical Inclusive OR	OR dest,src
OUT	Output to Port	OUT port,acc
OUTS	Output String	OUTS DX,src
POP	Pop Operand off Stack	POP dest
POPA	Pop All General Registers	POPA
POPF	Pop Flags off Stack	POPF
PUSH	Push Operand Onto Stack	PUSH src
PUSHA	Push All General Registers	PUSHA
PUSHF	Push Flags onto Stack	PUSHF
RCL	Rotate Left through Carry	RCL dest,count
RCR	Rotate Right through Carry	RCR dest,count
REP	Repeat	REP
REPE	Repeat while Equal	REPE
REPNE	Repeat while Not Equal	REPNE
REPNZ	Repeat while Not Zero	REPNZ
REPZ	Repeat while Zero	REPZ
RET	Return from Procedure	RET
ROL	Rotate Left	ROL dest,count
ROR	Rotate Right	ROR dest,count
SAHF	Store AH Register in Flags	SAHF
SAL	Shift Arithmetic Left	SAL dest,count
SAR	Shift Arithmetic Right	SAR dest,count
SBB	Subtract with Borrow	SBB dest,src

Table 2-3 (continued)
Complete 80386 Instruction Set

Instruction	Meaning	Assembler Format	
SCAS	Compare String	SCAS	dest
SETcc	Set Byte on Condition	SETcc	dest
SGDT	Store GDT Register	SGDT	dest
SHL	Shift Logical Left	SHL	dest,count
SHLD	Double Precision Shift Left	SHLD	dest,src,count
SHR	Shift Logical Right	SHR	dest,count
SHRD	Double Precision Shift Right	SHRD	dest,src,count
SIDT	Store IDT Register	SIDT	dest
SLDT	Store LDT Register	SLDT	dest
SMSW	Store Machine Status Word	SMSW	dest
STC	Set Carry Flag	STC	
STD	Set Direction Flag	STD	
STI	Set Interrupt Flag	STI	
STOS	Store String	STOS	dest
STR	Store Task Register	STR	dest
SUB	Subtract	SUB	dest,src
TEST	Logical Compare	TEST	dest,src
VERR	Verify Segment for Reading	VERR	sel
VERW	Verify Segment for Writing	VERW	sel
WAIT	Wait until BUSY# Negated	WAIT	
XCHG	Exchange Operand, Register	XCHG	dest,src
XLAT	Table Lookup	XLAT	source-table
XOR	Logical Exclusive OR	XOR	dest,src

* CWD sign extends register AX into registers DX and AX, whereas CWDE sign extends AX into EAX.

DEC	subtract 1
INC	add 1
NOT	logical NOT (complement or invert)
ROL	rotate left
ROR	rotate right
SBB	subtract with borrow
SHL	shift logical left
SHR	shift logical right
SUB	subtract
TEST	bit test
• Program Control	
CALL	call subroutine
INT	interrupt (trap)
JA	jump if above
JAE	jump if above or equal
JB	jump if below
JBE	jump if below or equal
JC	jump if carry
JE	jump if equal
JMP	jump unconditionally
JNC	jump if not carry
JNE	jump if not equal
JNS	jump if not sign
JNZ	jump if not zero
JS	jump if sign
JZ	jump if zero
RET	return from subroutine

Frequently Used Data Transfer Instructions

The frequently used data transfer instructions are IN, LEA, MOV, OUT, PUSH, and POP. Let us now describe them in more detail.

MOV moves data from one address to another. It is really a “copy” instruction, since the source does not change. The general form is

MOV destination,source

Note that the destination comes first. For example,

Table 2-4
Frequently Used 80386 Instructions

Instruction	Meaning
ADC	Add with carry
ADD	Add
AND	Logical AND
CALL	Call subroutine
CMP	Compare
DEC	Subtract 1
IN	Input
INC	Add 1
INT	Interrupt (trap)
JA	Jump if above
JAE	Jump if above or equal
JB	Jump if below
JBE	Jump if below or equal
JC	Jump if carry
JE	Jump if equal
JMP	Jump unconditionally
JNC	Jump if not carry
JNE	Jump if not equal
JNS	Jump if sign positive
JNZ	Jump if not zero
JS	Jump if sign negative
LEA	Load effective address
MOV	Move
NOT	Logical NOT (complement or invert)
OUT	Output
POP	Load from stack
PUSH	Store on stack
RET	Return from subroutine
ROL	Rotate left
ROR	Rotate right
SBB	Subtract with borrow

MOV BL,CL

moves the contents of register CL to BL. Register CL does not change. Be careful — the direction is the opposite of what you might expect. However, it is exactly the same as the direction in assignment statements (such as $C = B$) in high-level languages (BASIC, C, or Pascal). Reversing the source and destination in moves is a common error in 80386 assembly language programs.

Either operand in MOV can use any 80386 addressing mode. The only limitation is that one operand must be either a register or an immediate data value. Thus MOV can do the following transfers:

- Register to register
- Memory to register
- Register to memory
- Immediate data to register or memory

MOV can put immediate data into memory without using any registers. It cannot, however, transfer variable data from one memory address to another. Some examples are:

1. MOV AL,BL. This instruction moves a data byte from register BL to register AL. If a register is a move's source or destination, its length determines the size of the data (byte, word, or double word).
2. MOV AL,8[EBX]. This instruction loads a byte of data from an effective address into register AL. The effective address is the contents of EBX plus 8. Here again, the destination register (AL) determines the data's size.
3. MOV BYTE PTR [EBX+ESI],0CH. This instruction moves the value 0CH to the effective address given by the sum of registers EBX and ESI. As neither operand is a register, we need a way to indicate the data's size. The alternatives are:

BYTE PTR for 8-bit data

WORD PTR for 16-bit data

DWORD PTR for 32-bit data

Note that you must put one of these in the instruction. Most assemblers do not provide a default.

IN and OUT move data from and to peripherals, respectively. Peripherals have their own 64K address space, separate from memory addresses. We call this approach *isolated input/output*. I/O addresses are 16 bits long and are nonsegmented. That is, segment registers do not apply to I/O addresses, and hence no address translation is required.

IN and OUT have two primary forms:

IN accumulator, port address and OUT port address, accumulator transfer data to and from ports with addresses in the range 00-FF hex. Note, however, that Intel has reserved ports F8 through FF hex for use with coprocessors.

IN accumulator, DX and OUT DX, accumulator transfer data to and from ports addressed through register DX. This approach allows access to any address in the 64K I/O space.

Note that IN and OUT always use an accumulator (AL, AX, or EAX) and either a fixed port address or register DX. No other combinations are allowed. Note also that there are no brackets around DX, even though it is an indirect address.

Examples

1. The following instruction moves a byte from port 50 (hex) to accumulator AL:

```
IN    AL,50H
```

2. The following instruction sequence moves a double word from accumulator EAX to port number FF00 (hex):

```
MOV    DX,0FF00H
```

```
OUT    DX,EAX
```

3. The following instruction sequence loads EAX with a double word from port number C180 (hex):

```
MOV    DX,0C180H
```

```
IN     EAX,DX
```

PUSH and POP move data to and from the stack. They are usually 32-bit transfers. PUSH and POP are useful for saving registers during subroutine calls and other similar tasks. They are especially common in programs derived from high-level languages (such as Pascal) that transfer all parameters on the stack.

Examples

1. The following instruction stores the contents of register ECX at the top of the stack:

```
PUSH    ECX
```

2. The following instruction loads register ESI from the top of the stack:

```
POP     ESI
```


PUSH and POP both update the stack pointer automatically. PUSHAD and POPAD are special versions that move all general-purpose registers (see Figure 2-2) to and from the stack.

LEA loads an effective address into an index or base register. That is, it goes through the entire calculation specified by an addressing mode. But, instead of using the effective address, it simply saves it in a register. This is useful for computing an address parameter for a subroutine and for speeding up sequences that use the same effective address several times.

Examples

1. The following instruction loads register EBX with the contents of ESI plus 8:

```
LEA    EBX,8[ESI]
```

LEA is thus also useful for doing simple arithmetic, particularly because it does not overwrite the original operand. This single instruction is equivalent to

```
MOV    EBX,ESI
ADD    EBX,8
```

2. The following instruction loads register ECX with the sum of EBX, ESI, and 6:

```
LEA    ECX,6[EBX+ESI]
```

Not only does this avoid recalculations, but it also performs a step in sequences for multilevel indirect and indexed addressing.

3. The following instruction loads register EBX with 5 times the contents of EAX:

```
LEA    EBX,[EAX+4*EAX]
```

It uses scaled indexing to do a fast multiply. Of course, the process only works for multiplications by 2, 3, 4, 5, 8, or 9.

Frequently Used Arithmetic and Logical Instructions

The frequently used arithmetic and logical instructions are all straightforward. Note the following:

- Double-operand instructions take the form:

operation code destination,source

The result replaces the destination. For example:

```
SUB    EAX,EBX
```

subtracts the contents of register EBX from register EAX and puts the difference in EAX. The order here is what one would expect.

- CMP acts just like SUB but does not save the result. It affects only the flags.
- TEST acts just like a logical AND but does not save the result. It is thus the logical version of CMP.
- ADC and SBB include the carry in addition and subtraction, respectively. The results are:
ADC: $(\text{dest}) = (\text{dest}) + (\text{src}) + \text{Carry}$
SBB: $(\text{dest}) = (\text{dest}) - (\text{src}) - \text{Carry}$
- Shifts can specify their counts in any of three ways:
Implicitly for a single shift.
Immediate value (1 to 32).
Value in register CL (lowest 5 bits only). Note that only CL can be used in this way.

Typical forms are:

SHL AL,1 shifts AL left 1 bit position.

SHL AL, 5 shifts AL left 5 bit positions.

SHR AX,CL shifts AX right a number of bit positions given by the five least significant bits of register CL.

The 80386 has no explicit clear instruction. However, you can use

SUB reg,reg

to clear a register. The result is obviously zero. Many programmers prefer the equivalent but more confusing

XOR reg,reg

To verify that its result is zero, remember that the EXCLUSIVE OR of 2 bits is 0 if they are equal and 1 otherwise.

The results of arithmetic and logical instructions can end up in memory as well as in registers. For example:

1. The following instruction logically ANDs the 8-bit number F0 hex with the contents of address 4000 hex:

AND BYTE PTR [4000H],0F0H

2. The following instruction adds register EAX to the double word starting at the address 10 bytes beyond [EBX]:

ADD 10[EBX],EAX

3. The following instruction complements the 16-bit number at address 3000 hex:

NOT WORD PTR [3000H]

Frequently Used Program Control Instructions

The 80386's program control instructions are generally conventional. Conditional jumps can use only relative offsets, which may be 8, 16, or 32 bits long. CALL and JMP, on the other hand, can use any addressing mode, including relative. Note that jumps work like other instructions (unlike on many other processors, where they act as though one level of indirection had been removed). For example, JMP EBX transfers control to the address in EBX, whereas JMP [EBX] transfers control to the address reached indirectly via EBX.

The set of conditional jumps listed in Table 2-5 applies most often after a comparison of unsigned numbers. If op1 and op2 are unsigned, the jumps work as follows after CMP op1,op2:

JA (or JNBE)	jump if op1 > op2
JAE (or JNB)	jump if op1 ≥ op2
JB (or JNAE)	jump if op1 < op2
JBE (or JNA)	jump if op1 ≤ op2
JE	jump if op1 = op2
JNE	jump if op1 ≠ op2

The set listed in Table 2-6 applies most often after arithmetic or logical instructions. The jumps are:

JC	jump if Carry = 1
JNC	jump if Carry = 0
JNS	jump if positive (Sign = 0)
JNZ	jump if result not zero (Zero = 0)
JS	jump if negative (Sign = 1)
JZ	jump if result zero (Zero = 1)

Note that some mnemonics are just different names for the same instruction. For example, JB and JC are equivalent, as are JAE and JNC, JE and JZ, and JNE and JNZ. The alternative mnemonics serve simply to make programs clearer. The same holds for the more obvious JA and JNBE, JAE and JNB, JBE and JNA, and JB and JNAE.

INT (software interrupt or trap) is a special instruction that makes the processor do the following:

1. Save EFLAGS, the code segment register, and the instruction pointer at the top of the stack. EFLAGS is pushed first, then CS, and finally EIP.
2. Jump indirectly via an address determined from INT's parameter (an integer less than 256). We will discuss the derivation of the target address later. The target ad-

dress and its successors must contain new values for the instruction pointer and code segment register.

INT is used to respond to external interrupts. It also often allows user programs to access built-in routines in an operating system such as MS-DOS or in firmware such as a BIOS (Basic Input/Output System). For example, on an MS-DOS computer, INT 16H performs a keyboard operation such as reading a character, reporting whether a character is available, or obtaining the status of the shift key. For information on what INT instructions do in the IBM PC, see P. Norton, *Programmer's Guide to the IBM PC*, Microsoft Press, Redmond, WA, 1985.

General Data Transfer Instructions

Table 2-7 lists all 80386 data transfer instructions. Note the following:

1. XLAT does a simple table lookup. It computes an effective address by adding AL to EBX. It then moves a byte from the effective address to AL. AL's previous value is lost, but EBX is unaffected. This highly specialized instruction can handle any table with 8-bit elements.
2. There are several conversion instructions. They can do either sign- or zero-extended conversions of bytes to words and words to double words. They can also do sign-extended conversions of double words to quad words. Conversion instructions are particularly useful for extending counters, indexes, and dividends. They are also important in compilers for doing type conversions.

Examples

1. The following instruction exchanges the value in AL with the value in the byte addressed by register EBP:

XCHG AL,[EBP]

XCHG replaces three MOVs, since a temporary holding place would be necessary to avoid overwriting a value. That is, the alternative sequence using register BL would be

MOV	BL,AL	;SAVE FIRST OPERAND
MOV	AL,[EBP]	;MOVE SECOND OPERAND
MOV	[EBP],BL	;MOVE FIRST OPERAND

Table 2-5
Conditional Jumps After an Unsigned Comparison

Mnemonic	Description
JA	Jump if above (CF = 0 and ZF = 0)
JAЕ	Jump if above or equal (CF = 0)
JB	Jump if below (CF = 1)
JBE	Jump if below or equal (CF = 1 or ZF = 1)
JE	Jump if equal (ZF = 1)
JNE	Jump if not equal (ZF = 0)

Table 2-6
Conditional Jumps After Logical or Unsigned Arithmetic Instructions

JC	Jump if carry (CF = 1)
JNC	Jump if not carry (CF = 0)
JNS	Jump if not sign (SF = 0)
JNZ	Jump if not zero (ZF = 0)
JS	Jump if sign (SF = 1)
JZ	Jump if zero (ZF = 1)

2. The following instruction extends the value in AL to 32 bits with zeros and puts the extended value in EAX:

MOVZX EAX,AL

Bits 8 through 31 are all zero.

3. The following instruction converts the signed double word in EAX into a signed quad word in EDX and EAX:

CDQ

The more significant double word is in EDX. The extension propagates the most significant bit of EAX into all bits of EDX. No, CDQ are not the initials of a completely unknown descendant of Johann Sebastian Bach.

General Data Manipulation Instructions

Tables 2-8 through 2-11 list the 80386's arithmetic, string, logical, and bit manipulation instructions. String instructions (see Table 2-9), despite the name, are actually convenient for handling any data array, not just character strings. The idea is to make one iteration in an array processing sequence into an instruction. We call such instructions *string primitives*, as they are building blocks for complex string operations. In general, the instruction must:

1. Do an operation, such as loading, storing, comparing, or moving an element. Source and destination pointers define the addresses involved. Some operations, such as loading and storing, require only one address. Others, such as comparing or moving, require two.
2. Update the pointers to reach the next elements or available addresses. The sign of the updating step depends on whether we are moving up (*autoincrementing*) or down (*autodecrementing*) through the array. The size of the step depends on the size of the elements. It is 1 if the elements are 8 bits, 2 if they are 16 bits, and 4 if they are 32 bits.

The instruction specifies the size of the step, either through a suffix (B, W, or D) or through an operand. The D (direction) flag determines the sign. It is 0 for autoincrementing, 1 for autodecrementing.

One can make string instructions do even more by prefixing them with REP. It decrements a counter and repeats the operation if the result is not zero. The precise order of the steps is as follows:

1. Check whether register ECX contains zero and exit if it does. Nothing happens if ECX is zero initially.
2. Do the subsequent string instructions. REP works only with string instructions, not with moves or arithmetic or logical instructions.
3. Subtract 1 from ECX and return to step 1. The decrement does not affect the flags.

Conditional REPs (REPE, REPNE, REPNZ, or REPZ) repeat the string instruction only as long as their condition holds. For example, REPZ repeats only as long as the Zero flag is 1. The processor checks the condition after each iteration before decrementing ECX.

Note the following about other less frequently used data manipulation instructions:

Table 2-7
Data Transfer Instructions

Mnemonic	Assembler	Format	Flags										
			OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
GENERAL-PURPOSE													
MOV	MOV	dest,src	-	-	-	-	-	-	-	-	-	-	-
MOV	MOV	control, debug	U	U	U	U	U	U	-	-	-	-	-
POP	POP	dest	-	-	-	-	-	-	-	-	-	-	-
POPA	POPA		-	-	-	-	-	-	-	-	-	-	-
PUSH	PUSH	src	-	-	-	-	-	-	-	-	-	-	-
PUSHA	PUSHA		-	-	-	-	-	-	-	-	-	-	-
XCHG	XCHG	dest,src	-	-	-	-	-	-	-	-	-	-	-
XLAT	XLAT	source table	-	-	-	-	-	-	-	-	-	-	-
CONVERSION													
CBW	CBW		-	-	-	-	-	-	-	-	-	-	-
CDQ	CDQ		-	-	-	-	-	-	-	-	-	-	-
CWD	CWD		-	-	-	-	-	-	-	-	-	-	-
CWDE	CWDE		-	-	-	-	-	-	-	-	-	-	-
MOVSX	MOVSX	reg,src	-	-	-	-	-	-	-	-	-	-	-
MOVZX	MOVZX	reg,src	-	-	-	-	-	-	-	-	-	-	-
INPUT/OUTPUT													
IN	IN	acc,port	-	-	-	-	-	-	-	-	-	-	-
OUT	OUT	port,acc	-	-	-	-	-	-	-	-	-	-	-
ADDRESS OBJECT													
LDS	LDS	reg,src	-	-	-	-	-	-	-	-	-	-	-
LEA	LEA	reg,src	-	-	-	-	-	-	-	-	-	-	-

Table 2-7 (continued)
Data Transfer Instructions

Mnemonic	Assembler	Format	Flags										
			OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
LES	LES	reg,src	-	-	-	-	-	-	-	-	-	-	-
LFS	LFS	reg,src	-	-	-	-	-	-	-	-	-	-	-
LGS	LGS	reg,src	-	-	-	-	-	-	-	-	-	-	-
LSS	LSS	reg,src	-	-	-	-	-	-	-	-	-	-	-

FLAG MANIPULATION

CLC	CLC	-	-	-	-	-	0	-	-	-	-	-	-
CLD	CLD	-	-	-	-	-	-	-	-	0	-	-	-
CMC	CMC	-	-	-	-	-	C	-	-	-	-	-	-
LAHF	LAHF	-	-	-	-	-	-	-	-	-	-	-	-
POPF	POPF	R	R	R	R	R	R	R	R	R	R	R	R
POPFD	POPFD	R	R	R	R	R	R	R	R	R	R	R	R
PUSHF	PUSHF	-	-	-	-	-	-	-	-	-	-	-	-
PUSHFD	PUSHFD	-	-	-	-	-	-	-	-	-	-	-	-
SAHF	SAHF	-	R	R	R	R	R	-	-	-	-	-	-
STC	STC	-	-	-	-	-	1	-	-	-	-	-	-
STD	STD	-	-	-	-	-	-	-	-	1	-	-	-

Key to Codes:

- 0 = instruction clears flag
- 1 = instruction sets flag
- C = instruction complements (inverts) flag
- R = instruction restores prior value of flag
- U = instruction's effect on flag is undefined
- = instruction does not affect flag

1. DAA and DAS are special instructions for packed decimal addition and subtraction (that is, two digits per byte). They apply only to register AL and work only after an ADC, ADD, SBB, or SUB instruction.

Table 2-8.
Arithmetic Instructions

Mnemonic	Assembler Format		Flags										
			OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
ADDITION													
AAA	AAA		-	-	-	TM	-	M	-	-	-	-	-
ADC	ADC	dest,src	M	M	M	M	M	TM	-	-	-	-	-
ADD	ADD	dest,src	M	M	M	M	M	M	-	-	-	-	-
DAA	DAA		U	M	M	TM	M	TM	-	-	-	-	-
INC	INC	dest	M	M	M	M	M	-	-	-	-	-	-
SUBTRACTION													
AAS	AAS		-	-	-	TM	-	M	-	-	-	-	-
CMP	CMP	dest,src	M	M	M	M	M	M	-	-	-	-	-
DAS	DAS		U	M	M	TM	M	TM	-	-	-	-	-
DEC	DEC	dest	M	M	M	M	M	-	-	-	-	-	-
NEG	NEG	dest	M	M	M	M	M	M	-	-	-	-	-
SBB	SBB	dest,src	M	M	M	M	M	TM	-	-	-	-	-
SUB	SUB	dest,src	M	M	M	M	M	M	-	-	-	-	-
MULTIPLICATION													
AAM	AAM		-	M	M	-	M	-	-	-	-	-	-
IMUL	IMUL	acc,src	M	U	U	U	U	M	-	-	-	-	-
MUL	MUL	acc,src	M	U	U	U	U	M	-	-	-	-	-
DIVISION													
AAD	AAD		-	M	M	-	M	-	-	-	-	-	-
DIV	DIV	acc,src	U	U	U	U	U	U	-	-	-	-	-
IDIV	IDIV	acc,src	U	U	U	U	U	U	-	-	-	-	-

Table 2-8. (continued)
Arithmetic Instructions

Key to Codes:	M	=	instruction modifies flag (effect depends on operands)
	T	=	instruction tests flag
	U	=	instruction's effect on flag is undefined
	-	=	instruction does not affect flag

* The string primitives also have common formats with implied operands. These use a suffix of B, W, or D depending on the size of the data transfer (for example, CMPSB, LODSW, and SCASD).

2. AAA, AAD, AAM, and AAS are special instructions for unpacked decimal (ASCII) arithmetic (that is, one digit per byte). They apply only to register AX. AA is for "ASCII adjust," not to keep these instructions at the head of the alphabetical line.
3. Bit manipulation instructions (BT, BTC, BTR, and BTS) set the Carry flag from the tested bit. The bit number can be either an immediate constant or the contents of a general-purpose register.
4. Bit scan instructions (BSF and BSR) look for a 1 bit in the operand. They clear the Zero flag if none exists and set it otherwise. If they find a 1 bit, they return its index in the destination register. BSF (bit scan forward) starts the scan at bit 0, whereas BSR (bit scan reverse) starts the scan at the most significant bit. The MSB is bit 31 for 32-bit operations or bit 15 for 16-bit operations.

Examples

1. DAA. This converts an 8-bit binary sum in AL into a decimal sum, using the Carry and Auxiliary Carry flags. It works only after ADC AL or ADD AL.
2. MOVSD. This moves data from the address in ESI to the one in EDI. It then updates both ESI and EDI according to the D flag's value. The pointers are increased if D is 0 and decreased if it is 1. The step is always sufficient to reach the next element; that is, the step is 4 for MOVSD, 2 for MOVSW, and 1 for MOVSB. MOVS provides a memory-to-memory move. The data never occupies a user register.
3. REP STOSB. This sequence first checks whether ECX is 0. If it is, nothing happens. If not, the processor stores the contents of AL at the address in EDI. It then updates

Table 2-9
String Instructions

Mnemonic	Assembler Format*		Flags										
			OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
CMPS	CMPS	dest,src	M	M	M	M	M	M	-	-	T	-	-
INS	INS	dest,DX	-	-	-	-	-	-	-	-	T	-	-
LODS	LODS	src	-	-	-	-	-	-	-	-	T	-	-
MOVS	MOVS	dest,src	-	-	-	-	-	-	-	-	T	-	-
OUTS	OUTS	DX,src	-	-	-	-	-	-	-	-	T	-	-
REP	REP		-	-	-	-	-	-	-	-	-	-	-
REPE	REPE		-	-	T	-	-	-	-	-	-	-	-
REPNE	REPNE		-	-	T	-	-	-	-	-	-	-	-
REPNZ	REPZ		-	-	T	-	-	-	-	-	-	-	-
REPZ	REPZ		-	-	T	-	-	-	-	-	-	-	-
SCAS	SCAS	dest	M	M	M	M	M	M	-	-	T	-	-
STOS	STOS	dest	-	-	-	-	-	-	-	-	T	-	-

* The string primitives also have common formats with implied operands. These use a suffix of B, W, or D depending on the size of the data transfer (for example, CMPSB, LODSW, and SCASD).

Key to Codes: M = instruction modifies flag (effect depends on operands)
 T = instruction tests flag
 U = instruction's effect on flag is undefined
 - = instruction does not affect flag

EDI according to the D flag's value. The step is +1 if D is 0, -1 if D is 1. Finally, the processor subtracts 1 from ECX and starts the operation over again.

4. BTS EAX,4. This instruction sets the Carry flag from bit 4 of register EAX, then sets that bit to 1. The other flags are not affected. Note that all bit manipulation instructions test a bit. The differences among them are whether and how they change the bit afterward.

Table 2-10
Logical Instructions

Mnemonic	Assembler	Format	Flags										
			OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
LOGICALS													
AND	AND	dest,src	0	M	M	U	M	0	-	-	-	-	-
NOT	NOT	dest	-	-	-	-	-	-	-	-	-	-	-
OR	OR	dest,src	0	M	M	U	M	0	-	-	-	-	-
TEST	TEST	dest,src	0	M	M	U	M	0	-	-	-	-	-
XOR	XOR	dest,src	0	M	M	U	M	0	-	-	-	-	-
SHIFTS													
SAL 1	SAL	dest,1	M	M	M	U	M	M	-	-	-	-	-
SAL count	SAL	dest,count	U	M	M	U	M	M	-	-	-	-	-
SAR 1	SAR	dest,1	M	M	M	U	M	M	-	-	-	-	-
SAR count	SAR	dest,count	U	M	M	U	M	M	-	-	-	-	-
SHL 1	SHL	dest,1	M	M	M	U	M	M	-	-	-	-	-
SHL count	SHL	dest,count	U	M	M	U	M	M	-	-	-	-	-
SHLD	SHLD	dest, src,count	U	M	M	U	M	M	-	-	-	-	-
SHR 1	SHR	dest,1	M	M	M	U	M	M	-	-	-	-	-
SHR count	SHR	dest,count	U	M	M	U	M	M	-	-	-	-	-
SHRD	SHRD	dest, src,count	U	M	M	U	M	M	-	-	-	-	-
ROTATES													
RCL 1	RCL	dest,1	M	-	-	-	-	TM	-	-	-	-	-
RCL count	RCL	dest,count	U	-	-	-	-	TM	-	-	-	-	-
RCR 1	RCR	dest,1	M	-	-	-	-	TM	-	-	-	-	-
RCR count	RCR	dest,count	U	-	-	-	-	TM	-	-	-	-	-
ROL 1	ROL	dest,1	M	-	-	-	-	M	-	-	-	-	-
ROL count	ROL	dest,count	U	-	-	-	-	M	-	-	-	-	-

Table 2-10 (continued)
Logical Instructions

Mnemonic	Assembler	Format	Flags											
			OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF	
ROR 1	ROR	dest,1	M	-	-	-	-	M	-	-	-	-	-	
ROR count	ROR	dest,count	U	-	-	-	-	M	-	-	-	-	-	
Key to Codes:			M	=	instruction modifies flag (effect depends on operands)									
			T	=	instruction tests flag									
			U	=	instruction's effect on flag is undefined									
			-	=	instruction does not affect flag									
			0	=	instruction clears flag									

Table 2-11
Bit Manipulation Instructions

Mnemonic	Assembler	Format	Flags											
			OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF	
BSF	BSF	dest,src	U	U	M	U	U	U	-	-	-	-	-	
BSR	BSR	dest,src	U	U	M	U	U	U	-	-	-	-	-	
BT	BT	base,offset	U	U	U	U	U	M	-	-	-	-	-	
BTC	BTC	base,offset	U	U	U	U	U	M	-	-	-	-	-	
BTR	BTR	base,offset	U	U	U	U	U	M	-	-	-	-	-	
BTS	BTS	base,offset	U	U	U	U	U	M	-	-	-	-	-	
Key to Codes:			M	=	instruction modifies flag (effect depends on operands)									
			U	=	instruction's effect on flag is undefined									
			-	=	instruction does not affect flag									

General Program Control Instructions

Table 2-12 lists all 80386 program control instructions. Table 2-13 describes the conditional jumps in more detail. Note the following:

1. **LOOP** subtracts 1 from register **ECX**, then branches if the result is not zero. It is thus equivalent to the common **DEC ECX, JNE** sequence at the end of a loop. Only **ECX** can be used in this manner. Conditional **LOOPS** also check whether a condition holds before branching. That is, a conditional **LOOP** continues the iterations as long as the condition holds and **ECX** is nonzero.
2. **JECXZ** jumps if register **ECX** contains 0. It can thus test whether a conditional **LOOP** or **REP** continued through all its iterations (that is, reduced **ECX** to zero). A **JECXZ** after the loop will branch if **ECX** is zero and will continue otherwise. The nonzero condition means that the exit occurred before the normal end of the loop (that is, because the condition no longer held). The program may then have to complete the loop before proceeding. The existence of **LOOP** and **JECXZ** makes it preferable to use **ECX** as a counter whenever possible. **JECXZ** can also test whether a counter is zero before a loop begins. It can thus provide an immediate exit in case the loop should not be executed at all.
3. The **SETcc** instructions set a byte to 1 if the condition holds and to 0 if it does not. These instructions convert a condition into a stored value for later testing or use in boolean expressions in high-level languages. The conditions have the same mnemonics and meanings as described for conditional jumps in Table 2-13.

Other Instructions

Tables 2-14 through 2-16 list the other 80386 instructions. The categories are high-level language support (Table 2-14), protection model (Table 2-15), and processor control (Table 2-16). We will describe protection model instructions in Chapter 5. The high-level language instructions provide quick ways to check array bounds and assign and delete parameter blocks for procedure entries. The processor control instructions include **ESC**, used for instructions intended for the numerical coprocessor, and **LOCK**, which controls a signal used in multiprocessing applications.

Table 2-12
Program Control Instructions

Mnemonic	Assembler Format		Flags										
			OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
CONDITIONAL TRANSFERS													
JA	JA	dest	-	-	T	-	-	T	-	-	-	-	-
JAE	JAE	dest	-	-	-	-	-	T	-	-	-	-	-
JB	JB	dest	-	-	-	-	-	T	-	-	-	-	-
JBE	JBE	dest	-	-	T	-	-	T	-	-	-	-	-
JC	JC	dest	-	-	-	-	-	T	-	-	-	-	-
JCXZ	JCXZ	dest	-	-	-	-	-	-	-	-	-	-	-
JE	JE	dest	-	-	T	-	-	T	-	-	-	-	-
JECXZ	JECXZ	dest	-	-	-	-	-	-	-	-	-	-	-
JG	JG	dest	T	T	T	-	-	-	-	-	-	-	-
JGE	JGE	dest	T	T	-	-	-	-	-	-	-	-	-
JL	JL	dest	T	T	-	-	-	-	-	-	-	-	-
JLE	JLE	dest	T	T	T	-	-	-	-	-	-	-	-
JNA	JNA	dest	-	-	T	-	-	T	-	-	-	-	-
JNAE	JNAE	dest	-	-	-	-	-	T	-	-	-	-	-
JNB	JNB	dest	-	-	-	-	-	T	-	-	-	-	-
JNBE	JNBE	dest	-	-	T	-	-	T	-	-	-	-	-
JNC	JNC	dest	-	-	-	-	-	T	-	-	-	-	-
JNE	JNE	dest	-	-	T	-	-	-	-	-	-	-	-
JNG	JNG	dest	T	T	T	-	-	-	-	-	-	-	-
JNGE	JNGE	dest	T	T	-	-	-	-	-	-	-	-	-
JNL	JNL	dest	T	T	-	-	-	-	-	-	-	-	-
JNLE	JNLE	dest	T	T	T	-	-	-	-	-	-	-	-
JNO	JNO	dest	T	-	-	-	-	-	-	-	-	-	-
JNP	JNP	dest	-	-	-	-	T	-	-	-	-	-	-
JNS	JNS	dest	-	T	-	-	-	-	-	-	-	-	-
JNZ	JNZ	dest	-	-	T	-	-	-	-	-	-	-	-
JO	JO	dest	T	-	-	-	-	-	-	-	-	-	-
JP	JP	dest	-	-	-	-	T	-	-	-	-	-	-
JPE	JPE	dest	-	-	-	-	T	-	-	-	-	-	-

Table 2-12 (continued)
Program Control Instructions

Mnemonic	Assembler Format		Flags										
			OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
JPO	JPO	dest	-	-	-	-	T	-	-	-	-	-	-
JS	JS	dest	-	T	-	-	-	-	-	-	-	-	-
JZ	JZ	dest	-	-	T	-	-	-	-	-	-	-	-
SETA	SETA	dest	-	-	T	-	-	T	-	-	-	-	-
SETAE	SETAE	dest	-	-	-	-	-	T	-	-	-	-	-
SETB	SETB	dest	-	-	-	-	-	T	-	-	-	-	-
SETBE	SETBE	dest	-	-	T	-	-	T	-	-	-	-	-
SETC	SETC	dest	-	-	-	-	-	T	-	-	-	-	-
SETE	SETE	dest	-	-	T	-	-	-	-	-	-	-	-
SETG	SETG	dest	T	T	T	-	-	-	-	-	-	-	-
SETGE	SETGE	dest	T	T	-	-	-	-	-	-	-	-	-
SETL	SETL	dest	T	T	-	-	-	-	-	-	-	-	-
SETLE	SETLE	dest	T	T	T	-	-	-	-	-	-	-	-
SETNA	SETNA	dest	-	-	T	-	-	T	-	-	-	-	-
SETNAE	SETNAE	dest	-	-	-	-	-	T	-	-	-	-	-
SETNB	SETNB	dest	-	-	-	-	-	T	-	-	-	-	-
SETNBE	SETNBE	dest	-	-	T	-	-	T	-	-	-	-	-
SETNC	SETNC	dest	-	-	-	-	-	T	-	-	-	-	-
SETNE	SETNE	dest	-	-	T	-	-	-	-	-	-	-	-
SETNG	SETNG	dest	T	T	T	-	-	-	-	-	-	-	-
SETNGE	SETNGE	dest	T	T	-	-	-	-	-	-	-	-	-
SETNL	SETNL	dest	T	T	-	-	-	-	-	-	-	-	-
SETNLE	SETNLE	dest	T	T	T	-	-	-	-	-	-	-	-
SETNO	SETNO	dest	T	-	-	-	-	-	-	-	-	-	-
SETNP	SETNP	dest	-	-	-	-	T	-	-	-	-	-	-
SETNS	SETNS	dest	-	T	-	-	-	-	-	-	-	-	-
SETNZ	SETNZ	dest	-	-	T	-	-	-	-	-	-	-	-
SETO	SETO	dest	T	-	-	-	-	-	-	-	-	-	-
SETP	SETP	dest	-	-	-	-	T	-	-	-	-	-	-
SETPE	SETPE	dest	-	-	-	-	T	-	-	-	-	-	-
SETPO	SETPO	dest	-	-	-	-	T	-	-	-	-	-	-

Table 2-12 (continued)
Program Control Instructions

Mnemonic	Assembler Format		Flags										
			OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
SETS	SETS	dest	-	T	-	-	-	-	-	-	-	-	-
SETZ	SETZ	dest	-	-	T	-	-	-	-	-	-	-	-

UNCONDITIONAL TRANSFERS

CALL	CALL	dest	-	-	-	-	-	-	-	-	-	-	-
JMP	JMP	dest	-	-	-	-	-	-	-	-	-	-	-
RET	RET or RET num*		-	-	-	-	-	-	-	-	-	-	-

* num is the number of bytes popped from stack

ITERATION CONTROLS

LOOP	LOOP	dest	-	-	-	-	-	-	-	-	-	-	-
LOOPE	LOOPE	dest	-	-	T	-	-	-	-	-	-	-	-
LOOPNE	LOOPNE	dest	-	-	T	-	-	-	-	-	-	-	-
LOOPNZ	LOOPNZ	dest	-	-	T	-	-	-	-	-	-	-	-
LOOPZ	LOOPZ	dest	-	-	T	-	-	-	-	-	-	-	-

INTERRUPTS

CLI	CLI		-	-	-	-	-	-	-	0	-	-	-
INT	INT	inttype	-	-	-	-	-	-	0	0	-	0	-
INTO	INTO		-	-	-	-	-	-	0	0	-	0	-
IRET	IRET		R	R	R	R	R	R	R	R	R	T	-
STI	STI		-	-	-	-	-	-	-	1	-	-	-

Key to Codes:

- R = instruction restores prior value of flag
- T = instruction tests flag
- = instruction does not affect flag
- 0 = instruction resets flag
- 1 = instruction sets flag

ADDRESS AND OPERAND SIZE

Most 80386 instructions, as we have mentioned, can operate on 8-bit, 16-bit, or 32-bit operands. The ways of specifying lengths are:

- Through register operands
- Through operands derived from assembler variables with specific types
- Through explicit designations (type overrides) such as BYTE PTR, WORD PTR, and DWORD PTR

The differentiation between 16-bit and 32-bit operands and addresses requires more explanation. In fact, each code segment has a bit in its definition (or descriptor) that determines whether it is a 16-bit or a 32-bit segment. A 16-bit segment uses 16-bit addresses and operands unless otherwise specified. A 32-bit segment similarly uses 32-bit addresses and operands. The programmer assigns the address and operand size for each segment with a USE directive in an assembly language program.

You can override the segment specification for addresses or operands. That is, you can force a 16-bit segment to use 32-bit addresses, 32-bit operands, or both. And vice versa for a 32-bit segment. The usual way to do this is by just specifying the other length through a register operand or a PTR operator.

Note, however, that the assembler actually generates an address override or operand override prefix byte. Thus a 32-bit instruction in a 16-bit segment requires 1 or 2 override bytes. The same holds for a 16-bit instruction in a 32-bit segment.

Why use an override? Among the reasons are:

- To increase the speed or extend the addressing capability of 16-bit programs without rewriting them completely.
- To match the bit widths of I/O ports or channels.
- To save memory by using 16-bit arrays or tables rather than 32-bit.
- To agree with type definitions used in high-level language programs. Note that a common reason for using 16-bit data types in high-level languages (such as C and Pascal) is to save time and memory. Now, on the 80386, there is actually a penalty (the override bytes) for using 16-bit operands in a 32-bit segment.
- To interface correctly with programs written in high-level languages.

Some uses of overrides make sense only during the transitional period as 32-bit processors succeed 16-bit devices.

In fact, the 80386 does not have separate 8-, 16-, and 32-bit instructions. It has:

- 8-bit instructions
- Instructions conforming to the segment's type (16- or 32-bit)

Table 2-13
Conditional Jump Instructions

Instruction	Description	Condition (Jump if...)
JA	Jump If Above	CF = 0 and ZF = 0
JAE	Jump If Above or Equal	CF = 0
JB	Jump If Below	CF = 1
JBE	Jump If Below or Equal	CF = 1 or ZF = 1
JC	Jump If Carry	CF = 1
JCXZ	Jump If CX Is Zero	(CX) = 0
JE	Jump If Equal	ZF = 1
JECHZ	Jump If ECX Is Zero	(ECX) = 0
*JG	Jump If Greater	ZF = 0 and SF = OF
*JGE	Jump If Greater or Equal	SF = OF
*JL	Jump If Less	SF ≠ OF
*JLE	Jump If Less or Equal	ZF = 1 or SF ≠ OF
JNA	Jump If Not Above	CF = 1 or ZF = 1
JNAE	Jump If Not Above or Equal	CF = 1
JNB	Jump If Not Below	CF = 0
JNBE	Jump If Not Below or Equal	CF = 0 and ZF = 0
JNC	Jump If No Carry	CF = 0
JNE	Jump If Not Equal	ZF = 0
*JNG	Jump If Not Greater	ZF = 1 or SF ≠ OF
*JNGE	Jump If Not Greater or Equal	SF ≠ OF
*JNL	Jump If Not Less	SF = OF
*JNLE	Jump If Not Less or Equal	ZF = 0 and SF = OF
*JNO	Jump If No Overflow	OF = 0
JNP	Jump If Parity Odd	PF = 0
JNS	Jump If Sign Positive	SF = 0
JNZ	Jump If Not Zero	ZF = 0
*JO	Jump If Overflow	OF = 1
JP	Jump If Parity Even	PF = 1
JPE	Jump If Parity Even	PF = 1
JPO	Jump If Parity Odd	PF = 0

Table 2-13 (continued)
Conditional Jump Instructions

Instruction	Description	Condition (Jump if...)
JS	Jump If Sign Negative	SF = 1
JZ	Jump If Zero	ZF = 1

* Used mainly to deal with signed (two's complement) operands.

Note 1: The Parity flag is 1 if the parity of a byte is even and 0 if it is odd.

Note 2: The Sign flag is the most significant bit of the latest result. It is 1 if that result was a negative signed number and 0 if it was a positive signed number.

Table 2-14
High-Level Language Instructions

Mnemonic	Assembler	Format	Flags											
			OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF	
BOUND	BOUND	reg,bound	-	-	-	-	-	-	-	-	-	-	-	
ENTER	ENTER	storage, level	-	-	-	-	-	-	-	-	-	-	-	
LEAVE	LEAVE		-	-	-	-	-	-	-	-	-	-	-	

Key to Codes: - = instruction does not affect flag

Table 2-15
Protection Model Instructions

Mnemonic	Assembler Format	Flags											
		OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF	
ARPL	ARPL	sel,reg	-	-	M	-	-	-	-	-	-	-	-
CLTS	CLTS		-	-	-	-	-	-	-	-	-	-	-
LAR	LAR	reg,src	-	-	M	-	-	-	-	-	-	-	-
LGDT	LGDT	src	-	-	-	-	-	-	-	-	-	-	-
LIDT	LIDT	src	-	-	-	-	-	-	-	-	-	-	-
LLDT	LLDT	src	-	-	-	-	-	-	-	-	-	-	-
LMSW	LMSW	src	-	-	-	-	-	-	-	-	-	-	-
LSL	LSL	reg,src	-	-	M	-	-	-	-	-	-	-	-
LTR	LTR	src	-	-	-	-	-	-	-	-	-	-	-
SGDT	SGDT	dest	-	-	-	-	-	-	-	-	-	-	-
SIDT	SIDT	dest	-	-	-	-	-	-	-	-	-	-	-
SLDT	SLDT	dest	-	-	-	-	-	-	-	-	-	-	-
SMSW	SMSW	dest	-	-	-	-	-	-	-	-	-	-	-
STR	STR	dest	-	-	-	-	-	-	-	-	-	-	-
VERR	VERR	sel	-	-	M	-	-	-	-	-	-	-	-
VERW	VERW	sel	-	-	M	-	-	-	-	-	-	-	-

Key to Codes: M = instruction modifies flag (effect depends on operands)
 - = instruction does not affect flag

Table 2-16
Processor Control Instructions

Mnemonic	Assembler Format	Flags											
		OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF	
ESC	ESC	-	-	-	-	-	-	-	-	-	-	-	
HLT	HLT	-	-	-	-	-	-	-	-	-	-	-	
LOCK	LOCK	-	-	-	-	-	-	-	-	-	-	-	
NOP	NOP	-	-	-	-	-	-	-	-	-	-	-	
WAIT	WAIT	-	-	-	-	-	-	-	-	-	-	-	

- Overrides for address or operand length that can be applied to the conforming instructions

INSTRUCTION SPEEDUPS

Two major reasons for the 80386's improved performance over earlier processors are changes in its multiplication and shifting methods. The multiplication instructions use an early-out method. It recognizes when the rest of the multiplier is zero and quits. This is like noting that the problems

3106×25 and 3106×2500

are the same except for position. Previous processors, like a child just learning to multiply, continued through the zeros. The change increases speed greatly in common situations with small multipliers.

The 80386 does all shifts in a "barrel shifter." This device simply selects the proper output for each bit position rather than shifting the data one bit position at a time. The result is that all shifts take the same amount of time, regardless of how far they move the data. Multiple-bit shifts are common in arithmetic, communications, graphics, and signal processing applications.

INSTRUCTIONS AND FLAGS

80386 instructions have highly individualized effects on the flags. The only way to be sure of what happens is to use a reference such as Appendix B or Tables 2-7 through 2-16. Remember the following:

- Data transfer instructions such as MOV, IN, and OUT do not affect any flags. The common way to set the flags from a register's contents is with
`TEST reg,reg`
This does not affect the register. OR reg,reg and AND reg,reg are equivalent but not as obvious.
- Logical instructions, for reasons that escape me, always clear Carry except for NOT, which does not affect any flags at all.
- INC and DEC do not affect Carry. This allows their use in loops that do multiple-precision arithmetic. The Carry is needed to transfer a bit from one iteration to the next. To add or subtract 1 with an effect on Carry, use `ADD op1,1` or `SUB op1,1`.

- Bit test instructions affect only the Carry flag.
- Bit scan instructions affect only the Zero flag.
- Rotate instructions (RCL, RCR, ROL, and ROR) affect only the Carry and Overflow flags.
- Type conversion and sign extension instructions do not affect the flags.
- The decrementing of counters and updating of pointers in LOOP, REP, and string instructions do not affect the flags. CMPS and SCAS affect the flags only through the comparison; the other string instructions do not affect the flags.
- NOT (1's complement) does not affect the flags but NEG (2's complement) does.
- Divides (DIV, IDIV) do not affect the flags but multiplies (IMUL, MUL) do.

8086 AND 80286 COMPATIBILITY

Many software authors want their programs to run on a wide range of hardware. For example, most personal computer programs must run on 80386-, 80286-, and 8086/8088-based computers (that is, on advanced PCs, ATs, and standard PCs or across the IBM Personal System/2 line). It is then important to know which instructions work on all processors. Table 2-17 lists 80286 instructions that are not available on the 8086 and 8088 processors. We omit operation codes such as MOVSD and PUSHAD that are simply 32-bit versions of existing 16-bit instructions. Table 2-18 lists 80386 instructions that are not available on the 80286. Obviously, several levels of compatibility are possible here:

1. Programs that must run on 8086- or 8088-based computers cannot use any instructions from Tables 2-17 and 2-18. This applies to MS-DOS programs.
2. Programs that must run on 80286-based computers cannot use any instructions from Table 2-18. Such programs may not run on 8086/8088-based computers. This applies to OS/2 programs.

The only new addressing mode in the 80286 or 80386 is the 80386's scaled indexing. It lets the processor multiply the index by a factor of 2, 4, or 8. Note also that a few instructions (IMUL, PUSH, and POP) have an immediate mode on the 80286 and 80386 that does not exist on the 8086/8088.

There are other minor differences that are rare in practice. Here we will mention only those having to do with the ordinary instruction set rather than with segmentation or interrupts.

1. PUSH SP. This instruction saves the value of the stack pointer before incrementing it on the 80286 and 80386 but the value afterward on the 8086.
2. Shift and rotate counts. The 80286 and 80386 mask these counts to the low-order 5 bits, whereas the 8086 does not.
3. Dividing by largest negative number. The 80286 and 80386 can divide by the largest negative number, whereas the 8086 produces an exception.
4. Flags in stack. The setting of the flags in the stack differs slightly. On the 80286 and 80386, bits 12 through 14 are in use, whereas on the 8086, they are always 1s.

There are also the obvious physical and clocking differences. The 80386 runs faster than its predecessors and takes fewer clock cycles to do many instructions. It also has new instructions that replace undefined operation codes on the earlier devices.

ASSEMBLER DIRECTIVES

Before using 80386 instructions to write actual programs, we must introduce a few assembler directives or pseudo-operations. These identify procedures and assign places in memory to fixed data, instructions, and storage areas. We will not deal with directives that control segmentation or implement advanced features such as macros and conditional assembly. We have used notation from Microsoft's Macro Assembler. Most other assemblers are similar.

The common directives are:

DB	Define byte
DD	Define double word
DQ	Define quad word
DT	Define 10 bytes (for use with floating point numbers)
DW	Define word
END	End of program
EQU	Equate, define symbolic name

These are all standard operations. The data definition directives may specify an undefined initial value with the ? notation. This is useful for temporary storage rather than fixed data.

Table 2-17
80286 Instructions Not Available on 8086/8088 Processors

Operation Code	Meaning
ARPL	Adjust requested privilege level
BOUND	Detect value out of range
CLTS	Clear task switched flag
ENTER	Enter procedure
IMUL r/m,imm8	Immediate signed multiply
INS	Input string
LAR	Load access rights
LEAVE	Leave procedure
LGDT	Load global descriptor table register
LIDT	Load interrupt descriptor table register
LLDT	Load local descriptor table register
LMSW	Load machine status word
LSL	Load segment limit
LTR	Load task register
OUTS	Output string
POPA	Load all user registers from stack
PUSHA	Store all user registers on stack
PUSH IMMEDIATE	Store constant on stack
RCL r/m,imm8	Rotate left through carry by immediate count
RCR r/m,imm8	Rotate right through carry by immediate count
ROL r/m,imm8	Rotate left by immediate count
ROR r/m,imm8	Rotate right by immediate count
SAL r/m,imm8	Shift left arithmetic by immediate count
SAR r/m,imm8	Shift right arithmetic by immediate count
SHL r/m,imm8	Shift left logical by immediate count
SHR r/m,imm8	Shift right logical by immediate count
SGDT	Store global descriptor table register
SIDT	Store interrupt descriptor table register

Table 2-17 (continued)
80286 Instructions Not Available on 8086/8088 Processors

Operation Code	Meaning
SLDT	Store local descriptor table register
SMSW	Store machine status word
STR	Store task register
VERR	Verify read access
VERW	Verify write access

Table 2-18
80386 Instructions Not Available on the 80386 Processor

Operation Code	Meaning
BSF	Bit scan forward (look for first 1 bit)
BSR	Bit scan reverse (look for first 1 bit)
BT	Bit test
BTC	Bit test and complement
BTR	Bit test and reset (clear)
BTS	Bit test and set
CDQ	Convert dword to qword
CWDE	Convert word to dword sign extended
LFS	Load pointer in F segment register
LGS	Load pointer into G segment register
LSS	Load pointer into S (stack) segment register
MOVSX	Move byte to word or dword (or word to dword) sign extended
MOVZX	Move byte to word or dword (or word to dword) zero extended
SETcc	Set byte from condition code
SHLD	Shift double left
SHRD	Shift double right

We will also use the following operators besides the standard arithmetic symbols:

DUP Duplicate (with data definition directives)

OFFSET Offset, i.e., the number of bytes between an item and the beginning of the segment in which it is defined

PTR Specify type

The following are typical directives. Note that labels attached to them are not followed by a colon (for some unknown reason).

1. SCALE is the address of a byte in memory. Its initial value is 3:

```
SCALE DB 3
```

2. ERMSG is the address of the first byte of a memory area containing the ASCII characters for ERROR:

```
ERMSG DB 'ERROR'
```

3. COUNT is the address of the low byte of a memory area containing the hexadecimal number 3000. The high byte of the number is in address COUNT+1:

```
COUNT DW 3000H
```

4. SUBTBL is the address of the low byte of the first of four double words with undefined initial values. This creates a 16-byte temporary storage area:

```
SUBTBL DD 4 DUP (?)
```

5. The value of IRATE is the number 7:

```
IRATE EQU 7
```

Note that you must precede addresses with an OFFSET operator to refer to their values within a segment. For example:

```
MOV EBX,OFFSET BASE
```

This instruction loads register EBX with the offset of address BASE from the beginning of its segment.

SUMMARY

The 80386 has many general-purpose user registers. They include an accumulator EAX, a base register EBX, a count register ECX, a data register EDX, index registers EDI and ESI, a base pointer EBP, and a stack pointer ESP. Some registers (EAX, EBX, ECX, and EDX) are byte addressable, and all are word addressable to maintain compatibility with previous processors (8088, 8086, and 80286). The 80386 also has a flag

or status register and several special-purpose registers intended mainly for operating system use.

The 80836 can handle many different data types, including bits, bit fields, integers, decimal numbers, ASCII characters, and floating point and real numbers. Integers, the primary data type, can be 8 bits (bytes), 16 bits (words), or 32 bits (double words). Most instructions operate on integers.

The 80836 also has a wide variety of addressing modes. It allows register, immediate, direct, register indirect, and modes built up from combinations of bases, indexes, and displacements. One can also multiply the index by a scale factor that can be 2, 4, or 8.

The 80836's instruction set includes the usual data transfer, arithmetic, logical, program control, and status manipulation instructions. It also has conversion, bit manipulation, string, iteration control, and high-level language support instructions, as well as instructions specifically intended for protected multitasking operating systems. The major peculiarities are:

1. Moves take the form

MOV destination,source

in which the destination comes first.

2. Arithmetic and logical instructions take the form

operation code destination,source

The destination is the primary operand (the minuend in subtraction and comparison). It is also the place where the result is stored.

Assembly Language Programming

"Isn't that lovely?" she sighed. "It's my favorite program — fifteen minutes of silence — and after that there's a half hour of quiet and then an interlude of lull."

Norton Juster, *The Phantom Tollbooth*

When in doubt, use brute force.
Ken Thompson

Nobody should be allowed to program in assembly language.
Jim Isaak, *Computer Design*, Jan. 1, 1987.

This chapter describes assembly language programming for the 80386. With minor exceptions, this is the same as programming the 8086 and 80286. We start with simple programs and proceed through bit manipulation, shifts, decision making, array processing, table lookup, string manipulation, arithmetic, and data structure manipulation. Final sections discuss parameter passing methods, common programming errors, and

ways to make programs run faster. The emphasis here is on applications programming rather than on systems programming.

80386 HIGHLIGHTS

For the applications programmer, the 80386 has the following major new features:

- Bit manipulation instructions. They replace the logical instructions and shifts previously used for this purpose. The 80386 has bit scans as well as bit complement, reset, set, and test.
- Double-length shifts. SHLD and SHRD can rapidly shift unaligned data that occupies several memory locations or registers. The idea is to move bits from one unit to the next in a single operation.
- Any user register can be an index register (except ESP) or a base pointer. Compare this to previous processors in which only DI or SI could be index registers and only BP or BX could be base pointers. For example, on the 80386, you can use ECX or EAX as index registers or base pointers in addressing. This may make some register transfers and saving and restoring operations unnecessary. Be careful — the old restrictions still apply to 16-bit operations. Note, however, that many registers still have special uses, such as:

EAX for I/O data, multiplication and division, extension, table lookup, packed and unpacked decimal operations, and data in string instructions

EBX for table lookup

ECX as a loop, shift, or bit position counter

EDX for I/O addresses, multiplication, and division

ESI and EDI for pointers in string instructions

Thus you must still allocate registers carefully. Although EAX, ECX, and EDX may be usable as index registers or base pointers, they may not be available in practice.

- Scaled index addressing for handling arrays with multibyte elements. This mode saves a multiplication or shift when accessing elements that are 2, 4, or 8 bytes long.

The 80386 also shifts and multiplies much faster than earlier processors. A shift's execution time is independent of the number of bit positions shifted. Multiplication is much faster because of the early-out algorithm that recognizes when the remaining

multiplier is zero. The speedup makes the upper parts of registers readily available and reduces the advantage of using shifts and additions instead of multiplications.

Another change in the 80386 is the virtual elimination of the time penalty for address calculations. Increased pipelining means that such calculations occur in parallel with instruction fetch, instruction execution, and memory addressing. Programmers can therefore use complex addressing modes (including scaled indexing) freely.

The 80386 also has other new instructions that are occasionally useful (see Table 2-14). For example, the conversion instructions CDQ, CWDE, MOVSX, and MOVZX do both sign and zero extension. SETcc (set on condition code) is handy for creating 8-bit Boolean variables used in high-level languages such as C and Pascal.

SIMPLE PROGRAMS

Simple 80386 assembly language programs use the following features of the processor's instruction set:

- Moves can transfer data to or from a location defined by any addressing mode. The other operand must be either a register or immediate data.
- Logical and arithmetic instructions (addition, AND, comparison, EXCLUSIVE OR, OR, and subtraction) also must have one operand that is either a register or immediate data. The destination may be either a register or a memory location.
- An instruction may work on 8, 16, or 32 bits. Its length is set by the length of a source or destination register, by the type of a variable, or by a type override (BYTE PTR, WORD PTR, or DWORD PTR).

Examples

1. Logically OR registers AL and BL:

OR AL,BL

This is an 8-bit operation because AL and BL are 8-bit registers. The result ends up in AL.

2. Add register EBX to register EAX:

ADD EAX,EBX

This is a 32-bit operation because EAX and EBX are 32-bit registers. The sum ends up in EAX.

3. Logically AND register AL with the binary constant BICON:

AND AL,BICON

BICON must be an 8-bit constant. Immediate addressing is the default mode. No special operation code or designator is necessary. TEST is the same as AND except that it does not change the destination.

4. Logically OR register AL with the data at the address in register EBX:

OR AL,[EBX]

The brackets around EBX indicate that it contains an address, not data.

5. Add contents of memory locations OPER1 and OPER2, put sum in memory location SUM:

```
MOV    AL,[OPER1]           ;GET FIRST OPERAND
ADD    AL,[OPER2]           ;ADD SECOND OPERAND
MOV    [SUM],AL             ;SAVE SUM
```

There are no memory-to-memory operations.

We can also use displacements from a base address. That is,

```
MOV    EBX,OPER1            ;POINT TO BASE ADDRESS
MOV    AL,[EBX]             ;GET FIRST OPERAND
ADD    AL,OPER2-OPER1[EBX]  ;ADD SECOND OPERAND
MOV    SUM-OPER1[EBX],AL    ;SAVE SUM
```

This is advantageous only if the locations are close together or if the entire data area could be moved as a unit.

6. Add a constant (VALUE) to the double word at address OPER:

ADD DWORD PTR VALUE,[OPER]

As neither operand is a register, we need DWORD PTR to indicate a 32-bit operation. Note that results can go directly into memory.

You can use INC and DEC to add and subtract 1. They can work on memory directly, but you need a typed variable or PTR to indicate the operation's length. Remember that neither INC nor DEC affects Carry.

7. Subtract 1 from the contents of the memory location 5 bytes beyond the address in EBX:

DEC BYTE PTR 5[EBX]

8. Add 1 to the double word at location ADDR:

INC DWORD PTR [ADDR]

The only way to recognize a Carry is by examining the Zero flag.

BIT MANIPULATION

The 80386 has special bit test, set, clear (reset), and complement (invert) instructions. They all first set the Carry flag from the bit value. The other flags are not affected. The bit number can be an immediate value or the contents of a register. Bit manipulation instructions apply only to 16- or 32-bit operands. There are no special 8-bit forms. The 80386 also has instructions for finding the first 1 bit in a word or double word.

Examples

1. Set bit 6 of register EDX:

`BTS EDX,6`

2. Clear bit 3 of register EAX:

`BTR EAX,3`

The clear instruction is BTR (R for “reset”).

3. Invert (complement) bit 14 of the double word starting at location ADDR:

`BTC [ADDR],14`

C is for “complement,” not “clear.” Bit manipulation instructions can operate on words or double words in memory as well as on 16-bit or 32-bit registers.

4. Test bit 5 of register EAX. That is, set the Carry flag if bit 5 is 1, and clear it if bit 5 is 0:

`BT EAX,5`

Note that all bit manipulation instructions do a test. Thus they all change the Carry as well. Watch for this when using BTC, BTR, or BTS.

You can use logical instructions to do more complex bit manipulation (Boolean) operations as follows:

- To set bits, logically OR them with 1s in the required positions.
- To clear bits, logically AND them with 0s in the required positions.
- To complement (invert) bits, logically EXCLUSIVE OR them with 1s in the required positions.
- To test bits (for all 0s), logically AND them with 1s in the required positions.

This approach lets you change or test several bits with one instruction. Of course, the changes must all be the same type. That is, you cannot set some bits and clear others.

Examples

1. Set bits 2 and 3 of register AL:

OR AL,00001100B

This single instruction replaces two BTS instructions. Besides, logical instructions can operate on bytes as well as on words and double words.

2. Clear bits 13, 14, and 15 of register ECX:

AND ECX,0FFFF1FFFH

This replaces three BTR instructions. Note that it changes all flags, whereas BTR affects only Carry.

3. Invert bits 1, 5, and 7 of the byte at address ADDR:

XOR BYTE PTR [ADDR],10100010B

This replaces three BTC instructions. Note that the bit positions can be scattered anywhere. XOR can also determine the bit positions in which two numbers differ.

For example,

XOR AL,BL

results in a 1 in each bit position in which AL and BL differ.

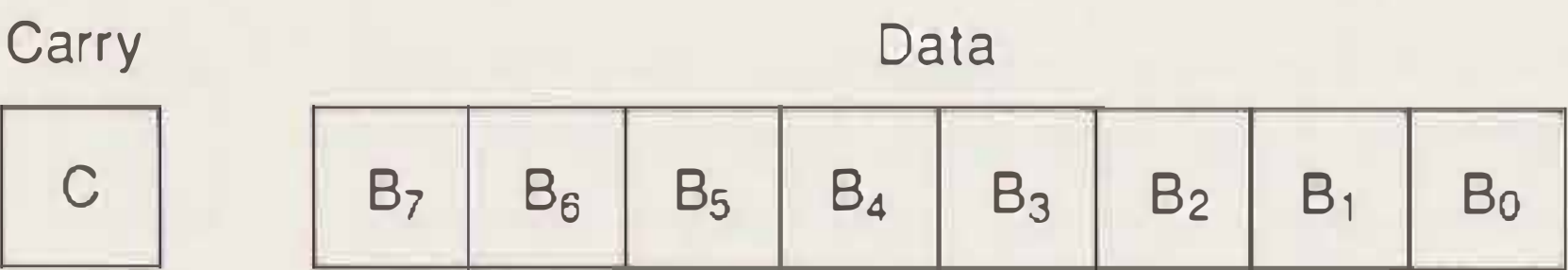
A handy shortcut to changing bit 0 of a register or memory location is to use INC or DEC. INC sets a bit if you know that it is cleared; DEC clears a bit if you know that it is set. Also either complements bit 0 if you are not using the other bits. This approach works if you have used SETcc to give the location a value of 0 or 1.

SHIFT OPERATIONS

The 80386 has a wide variety of shift instructions. They can work on any register or memory location. The number of bits to be shifted can be either an immediate constant or the contents of register CL. The instructions are:

RCL	(rotate left through carry)
RCR	(rotate right through carry)
ROL	(rotate left)
ROR	(rotate right)
SAR	(shift right arithmetic)
SHL	(shift left logical)
SHLD	(double-precision shift left)

Original contents of Carry flag and register or memory location



After RCL (rotate left through Carry)

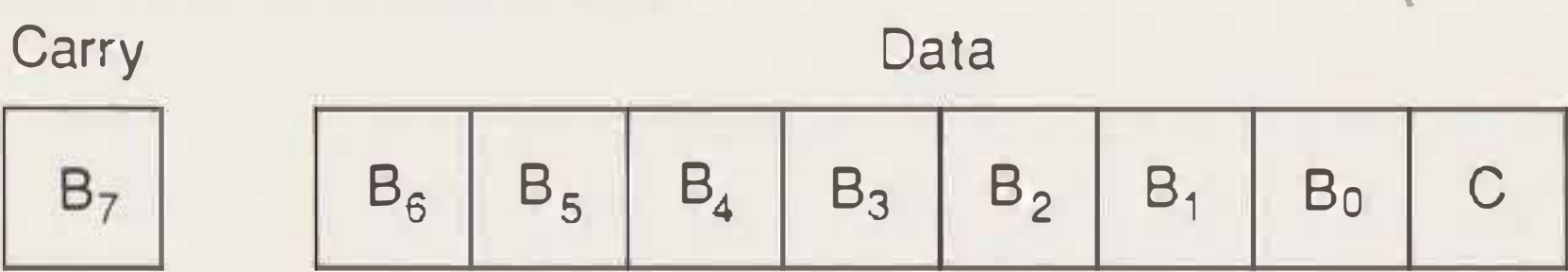
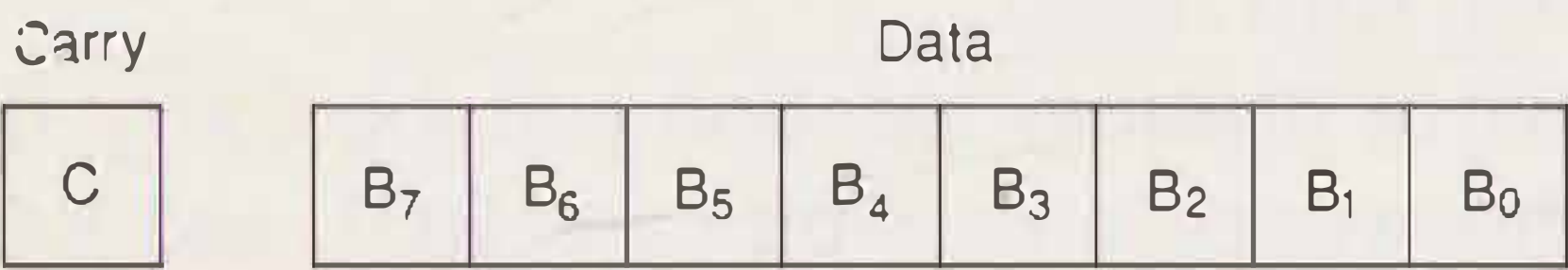


Figure 3-1
The RCL (rotate left through carry) instruction in its byte-length form.

Original contents of Carry flag and register or memory location



After RCR (rotate right through Carry)

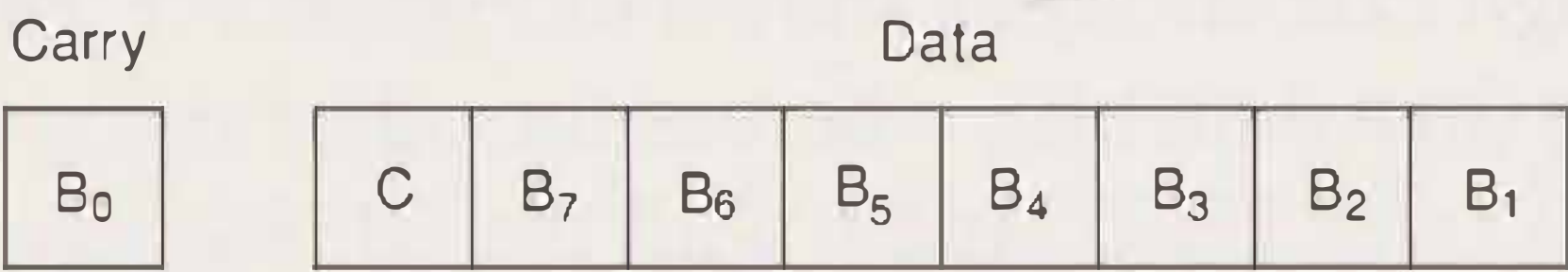
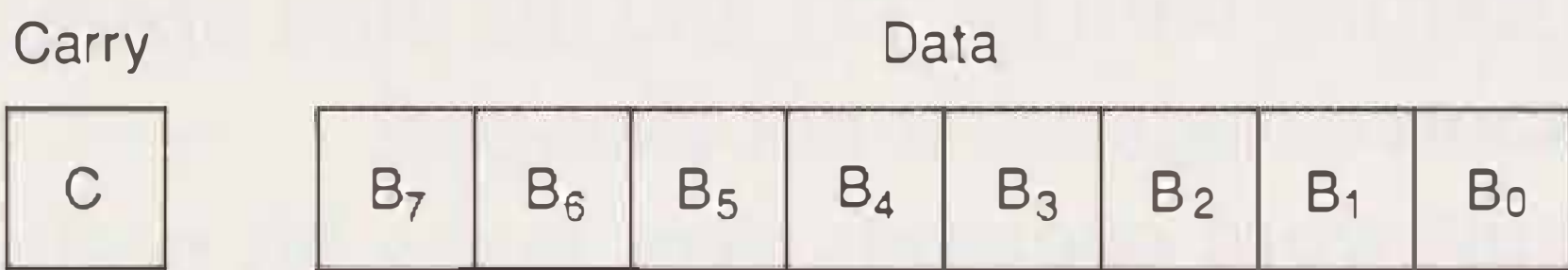


Figure 3-2
The RCR (rotate right through carry) instruction in its byte-length form.

Original contents of Carry flag and register or memory location



After ROL (rotate left)

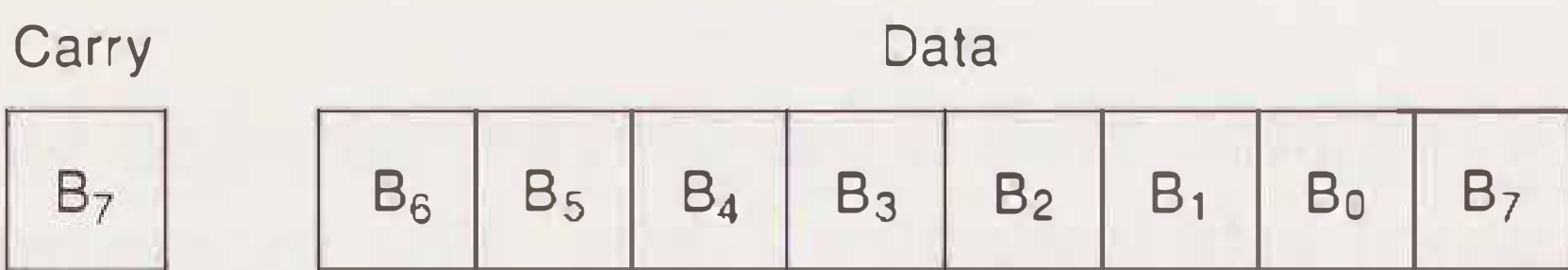
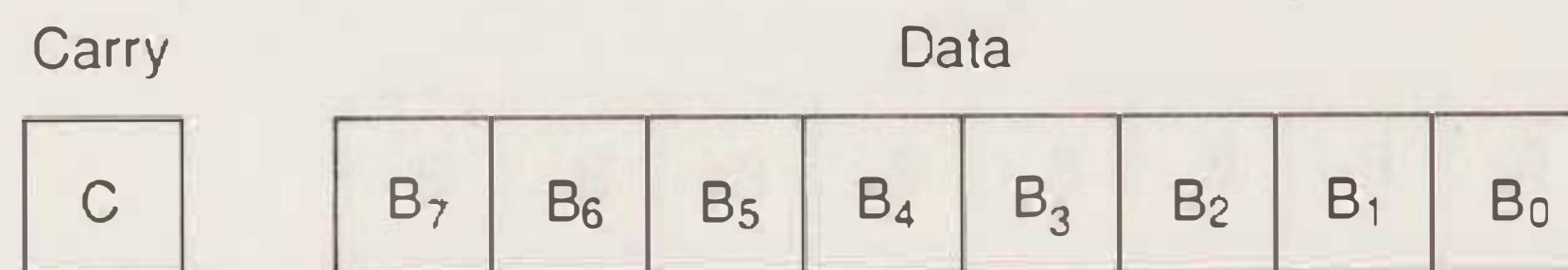
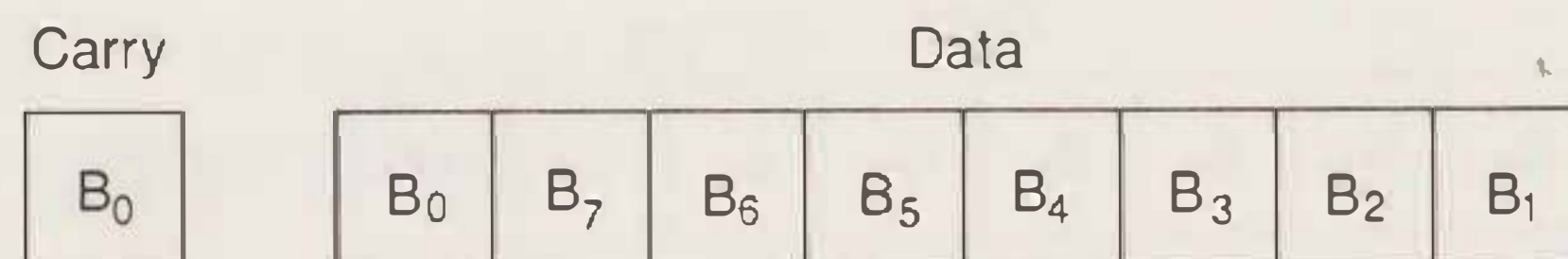


Figure 3-3
The ROL (rotate left) instruction in its byte-length form.

Original contents of Carry flag and register or memory location



After ROR (rotate right)

**Figure 3-4**

The ROR (rotate right) instruction in its byte-length form.

SHR (shift right logical)

SHRD (double-precision shift right)

RCL and RCR rotate a register or memory location and the Carry flag as a unit. Figures 3-1 and 3-2 show how this works in an 8-bit case. ROL and ROR rotate the register or memory location alone. The bit shifted off the end still appears in the Carry flag as well as at the other end. Figures 3-3 and 3-4 illustrate this. SHL (or SAL) and SHR are logical shifts that fill the vacated bits with 0s (see Figures 3-5 and 3-6). SAR copies the sign bit to the right (*sign extension*) as shown in Figure 3-7. Note that RCL and RCR preserve the Carry flag (in a data bit), whereas the other shifts destroy it.

The double-length shifts SHLD and SHRD (new with the 80386) have the following 32-bit forms:

SHLD r/m32, r32, imm8

SHLD r/m32, r32, CL

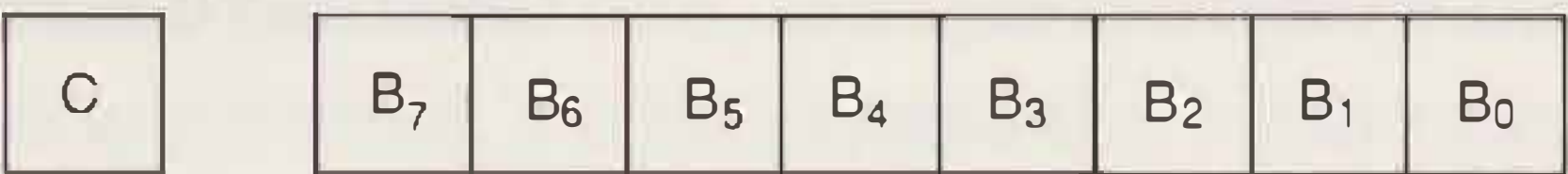
The first operand is the register or memory location to be shifted, the second operand contains the bits to be shifted in (starting with bit 31), and the third operand is the shift count. The second operand (the source register) is not changed.

SHLD and SHRD can shift multiword operands over many bit positions. For example, say we have an operand stored starting with its low byte at location ADDR. Its length in double words is COUNT, and we want to shift it left logically 4 bits. The goal is to move 4 bits quickly from one double word to the next.

The procedure is as follows:

1. Move the low double word to a register. Call it the previous double word.
2. Shift the original low double word left logically 4 bits. The incoming bits are all zeros here.

Original contents of Carry flag and register or memory location



After SHL (shift logical left)

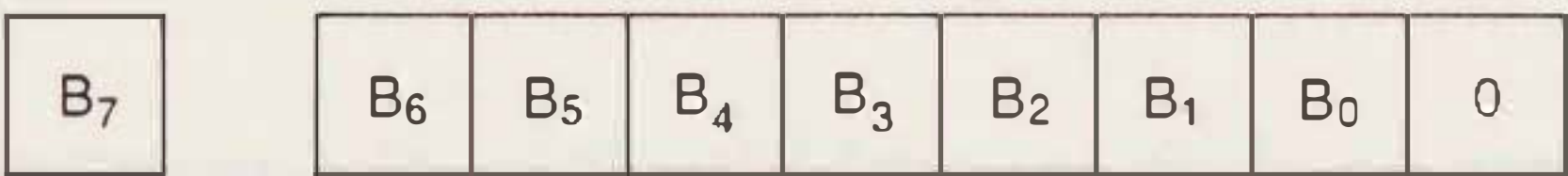
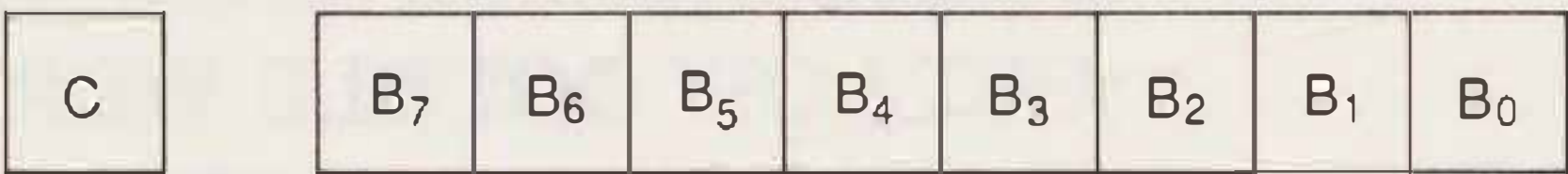


Figure 3-5
The SHL (shift logical left) or SAL (shift arithmetic left) instruction in its byte-length form.

Original contents of Carry flag and register or memory location

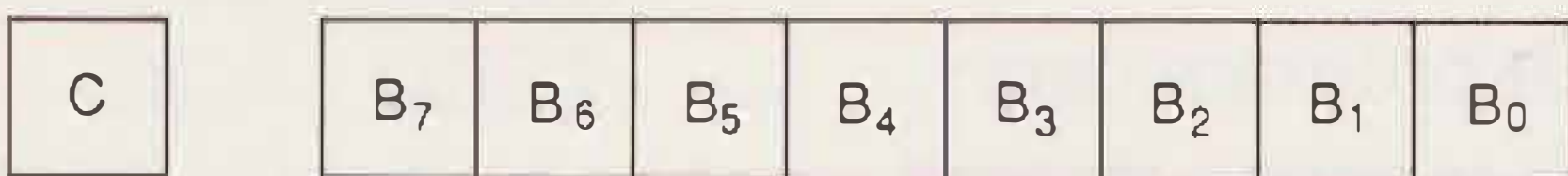


After SHR (shift logical right)



Figure 3-6
The SHR (shift logical right) instruction in its byte-length form.

Original contents of Carry flag and register or memory location



After SAR (shift arithmetic right)

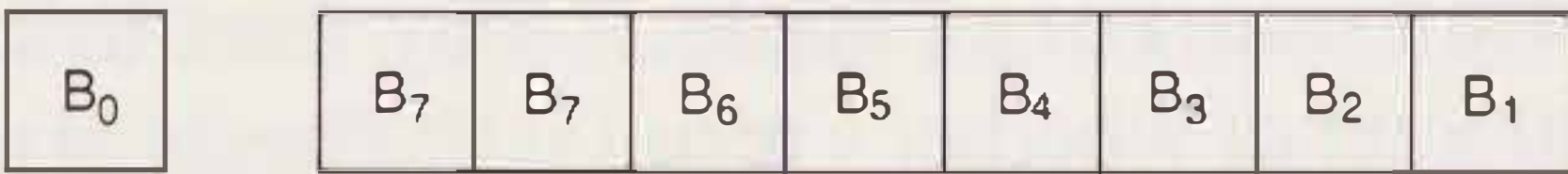


Figure 3-7
The SAR (shift arithmetic right) instruction in its byte-length form.

3. Move the next higher double word to a register. We must save it before shifting it for use in the next shift.
4. Use SHLD to shift the original next higher double word left 4 bits, deriving the incoming bits from the previous double word.
5. Move the original next higher double word to the register that held the previous double word.
6. Repeat steps 3 through 5 until the entire operand has been shifted.

SHLD's value here is that it lets us shift several bits from the previous double word into the current double word. The program is

```
MOV ESI,ADDR      ;POINT TO LOW DOUBLE WORD
MOV ECX,COUNT-1   ;GET NUMBER OF DOUBLE WORDS
                  ; AFTER FIRST
CLD               ;SELECT AUTOINCREMENTING
LODSD             ;GET LOW DOUBLE WORD
SHL  -4[ESI],4    ;SHIFT LOW DOUBLE WORD
ROTDW: MOV EBX,EAX ;SAVE PREVIOUS DOUBLE WORD
        LODSD     ;GET CURRENT DOUBLE WORD
        SHLD  -4[ESI],EBX,4;ROTATE CURRENT DOUBLE WORD
        LOOP  ROTDW ;COUNT DOUBLE WORDS
```

This kind of operation is useful in performing multiple-precision arithmetic, moving graphics figures, and examining bit patterns for communications applications.

MAKING DECISIONS

The major types of decisions in programs are:

- Deciding whether a bit is set or cleared
- Deciding whether two values are equal
- Deciding whether one value is greater than or less than another

The first type of decision lets the processor test the value of a flag, switch, status line, or other binary (ON/OFF) input. The second type lets the processor check whether an input or a result has a specific value. For example, it may want to know whether a keyboard input is a specific character or whether a result is 0. The third type of decision lets the processor determine whether a value is valid or is above or below a threshold, warning level, or set point.

Examples

1. Branch to DEST if AL contains the number VALUE:

```
CMP    AL,VALUE    ;DOES AL CONTAIN VALUE?
JE     DEST        ;YES, BRANCH
```

VALUE is a number as it has no special designator.

2. Branch to DEST if AL's contents are not the same as those of memory location ADDR:

```
CMP    AL,[ADDR]   ;IS AL SAME AS DATA IN MEMORY?
JNE    DEST        ;NO, BRANCH
```

3. Branch to DEST if EAX contains 0:

```
TEST   EAX,EAX     ;SET FLAGS FROM EAX
JZ     DEST        ;BRANCH IF EAX CONTAINS ZERO
```

Zero is a special value. No comparison is necessary. OR EAX,EAX and AND EAX,EAX have the same effect as TEST EAX,EAX but are not as obvious.

4. Branch to DEST if ECX does not contain -1 (FFFFFFFF hex):

```
INC    ECX         ;ESTABLISH ZERO FLAG
JNZ    DEST        ;BRANCH IF ECX WAS NOT -1
```

+1 and -1 are also special values. Remember that INC does not affect Carry.

5. Branch to DEST if EBX contains 1:

```
DEC    EBX         ;ESTABLISH ZERO FLAG
JZ     DEST        ;BRANCH IF EBX WAS 1
```

6. Branch to DEST if memory location ADDR contains 0:

```
MOV    AL,[ADDR]   ;GET VALUE
TEST   AL,AL       ;SET FLAGS
JZ     DEST        ;BRANCH IF VALUE WAS 0
```

MOV does not affect the flags. An alternative that does not use a register is

```
INC    BYTE PTR [ADDR] ;ESTABLISH ZERO FLAG IN
DEC    BYTE PTR [ADDR] ; TWO STEPS
JZ     DEST        ;BRANCH IF VALUE WAS 0
```

ADDR's contents are unchanged.

7. Branch to DEST if the double word at address ADDR contains the value VAL32:

```
CMP    DWORD PTR [ADDR],VAL32
JE     DEST
```

The destination can be in memory as long as the source is either a register or an immediate value. Of course, you must indicate the amount of data being handled with a typed variable or a PTR operator.

The way to determine how two operands compare in magnitude is to use CMP. If, as is usual, the operands are unsigned, the Carry flag indicates which is larger. Its value is

- 1 if the source is larger than the destination (that is, a borrow is necessary)
- 0 if the source is less than or equal to the destination

The unsigned conditional jumps are:

JC (JB) — jump if carry (below), jump if the destination is less than the source

JBE (JNA) — jump if below or equal (not above), jump if the destination is less than or equal to the source

JNC (JNB) — jump if no carry (not below), jump if the destination is greater than or equal to the source

JA (JNBE) — jump if above (not below or equal), jump if the destination is greater than the source

Examples

1. Jump to DEST if AL's contents are greater than or equal to the number VALUE:

```
CMP    AL,VALUE        ;IS AL ABOVE VALUE?
JAE     DEST            ;YES, BRANCH
```

The jump occurs if no borrow is necessary.

2. Jump to DEST if the double word at address OPER1 is less than the one at address OPER2:

```
MOV     EAX,[OPER1]     ;GET FIRST OPERAND
CMP     EAX,[OPER2]     ;IS SECOND OPERAND GREATER?
JB      DEST            ;YES, BRANCH
```

We can also use the mnemonic JB. It is equivalent to JC but more descriptive. Note that CMP cannot compare two direct addresses.

3. Jump to DEST if the double word at address OPER1 is less than or equal to the one at address OPER2:

```
MOV     EAX,[OPER1]     ;GET FIRST OPERAND
CMP     EAX,[OPER2]     ;IS SECOND OPERAND GREATER
                        ; OR SAME?
JBE     DEST            ;YES, BRANCH
```

4. Jump to DEST if the contents of register EDI are greater than or equal to VAL32:

```
CMP     EDI,VAL32        ;IS EDI ABOVE VAL32?
```


JAE DEST ;YES, BRANCH

CMP can use any register, but there are often special short forms for the accumulator.

With signed operands, we must account for 2's complement overflow. This is the case in which the difference between the operands affects the sign bit. That is, the result is outside the signed range for the specified number of bits.

If overflow is possible, we must use signed conditional jumps. These are:

JG (JNLE) — jump if greater, jump if destination is greater than source in the signed sense

JGE (JNL) — jump if greater or equal, jump if destination is greater than or equal to source in the signed sense

JLE (JNG) — jump if less or equal, jump if destination is less than or equal to source in the signed sense

JL (JNGE) — jump if less, jump if destination is less than source in the signed sense
These instructions test for overflow automatically.

Examples

1. Jump to DEST if AL's signed contents are greater than or equal to the signed number VALUE:

```
CMP    AL,VALUE           ;IS AL ABOVE VALUE?
JGE     DEST              ;YES, BRANCH
```

2. Jump to DEST if the signed double word at address OPER1 is less than the one at address OPER2:

```
MOV     EAX,[OPER1]       ;GET FIRST OPERAND
CMP     EAX,[OPER2]       ;IS SECOND OPERAND GREATER?
JL      DEST              ;YES, BRANCH
```

3. Jump to DEST if the signed double word at address OPER1 is less than or equal to the one at address OPER2:

```
MOV     EAX,[OPER1]       ;GET FIRST OPERAND
CMP     EAX,[OPER2]       ;IS SECOND OPERAND GREATER
                        ; OR SAME?
JLE     DEST              ;YES, BRANCH
```

LOOPING

The simplest way to repeat a sequence of instructions with the 80386 is as follows:

1. Load register ECX with the number of repetitions.
2. Do the sequence.
3. Use the LOOP instruction to subtract 1 from ECX and return to step 2 if the result is non-zero.

LOOP is handy because it combines a decrement and a conditional jump. Note that it always uses register ECX. The LOOPZ (LOOPE) and LOOPNZ (LOOPNE) variations also exit if their condition is not satisfied.

Typical programs have the following structure:

```
        MOV  ECX,NTIMES;NTIMES = NUMBER OF ITERATIONS
START:  Instructions to be repeated
        LOOP START    ;COUNT ITERATIONS
```

You can put JECXZ EXIT after the MOV to exit immediately if NTIMES is zero. This eliminates a potential source of error at a small cost.

By default, LOOP requires an 8-bit relative offset. However, you can use DWORD to override the default and allow a 16-bit relative offset. The instruction would then be

```
        LOOP  DWORD START    ;COUNT ITERATIONS
```

We can, of course, use other registers for counting or count up rather than down. These alternatives require different initializations, explicit INC or DEC instructions, and a JNZ at the end. In any case, the instructions to be repeated must not interfere with counting the iterations. Note that register ECX is special, and most programmers reserve it as a loop counter.

ARRAY MANIPULATION

The easiest way to access a particular element of an array is by placing its address in an index or base register. In this way, we can:

- Manipulate the element by referring to it indirectly, that is, as [reg].
- Access nearby elements with appropriate displacements, that is, as disp[reg].

Examples

1. Add an 8-bit element of an array to AL. Assume that the element's address is in register EBX. Add 1 to EBX afterward so that it contains the address of the next 8-bit element:

```

      ADD    AL,[EBX]          ;ADD CURRENT ELEMENT
      INC    EBX              ;POINT TO NEXT ELEMENT

```

The procedure is the same for 16-bit and 32-bit elements, except that more INCs or an ADD are necessary. Note that the 80386 has no explicit autoincrementing or autodecrementing.

2. Load EAX with the thirty-fifth element from an array of double words. Assume that the array's base address is in register EBX:

```

      MOV    ESI,35            ;GET OFFSET FOR ELEMENT
      MOV    EAX,[EBX+4*ESI]   ;OBTAIN ELEMENT

```

Scaled indexing is useful for 16-, 32-, and 64-bit elements. Remember that the scaling factor can only be 2, 4, or 8.

3. Exchange an element of an array with its successor if the two are not already in descending order. Assume that the elements are 8-bit unsigned numbers and that the address of the current element is in register EBX. Make EBX contain the address of the successor element afterward:

```

      MOV    AL,[EBX]          ;GET CURRENT ELEMENT
      CMP    AL,1[EBX]         ;COMPARE TO SUCCESSOR
      JAE    DONE              ;DONE IF IN ORDER
      XCHG   AL,1[EBX]         ;REPLACE SUCCESSOR
      MOV    [EBX],AL          ;REPLACE CURRENT ELEMENT
DONE: INC    EBX               ;UPDATE POINTER

```

This type of operation is useful in exchange sorts.

An important task in array manipulation is bounds checking. It involves determining whether an index is valid, that is, above or equal to a lower bound and below or equal to an upper bound. Bounds checking ensures that the processor only reads or writes valid elements. Otherwise, an array operation could refer accidentally to unrelated data.

The 80386 has a special instruction BOUND for bounds checking. Its form is

```
BOUND r32,m32
```

The result is a system jump (called a *trap* and discussed in Chapter 7) if $r32 < [m32]$ or $r32 > [m32+4]$. That is, a special event (or *exception*) occurs if the index is below

the lower bound or above the upper bound. The lower bound is at the specified address and the upper bound comes immediately afterward. The trap is interrupt 5 (see Chapter 7). Note that the index and the bounds can be in any units (bytes, words, or double words).

In practice, designers usually put the bounds just ahead of the array itself. This increases the array's storage requirements by 8 bytes.

TABLE LOOKUP

If the table's position in memory is fixed, we can access it by using its base address as a displacement. The element number then goes in an index or base register. Scaled indexing lets us access tables with multibyte elements.

If the table's position is not fixed, its base address should go in a base register and the element number in an index register. The special instruction XLAT (or XLATB) can access a table with 8-bit indexes and elements.

Examples

1. Load register AL with an element from a table. Assume that the table starts at BASE and the index is in memory location INDEX:

```
MOV    AL,[INDEX]        ;GET 8-BIT INDEX
MOV    EBX,BASE           ;GET BASE ADDRESS
XLAT                    ;GET 8-BIT ELEMENT
```

XLAT (translate) adds AL and EBX, then uses the sum as the address from which to load AL. This highly specialized instruction thus does a table lookup, assuming an 8-bit index and 8-bit elements. EBX is unaffected, so it can be used again later, but the index in AL is destroyed. Note that we do not have to extend AL to 32 bits explicitly.

2. Load EAX with an element from a table. Assume that its base address is BASE (a constant) and the index is in locations INDEX through INDEX+3:

```
MOV    ESI,[INDEX]        ;GET 32-BIT INDEX
MOV    EAX,BASE[ESI*4]     ;GET 32-BIT ELEMENT
```

Scaled indexing is convenient for handling multibyte elements.

3. Load EBX with an element from a table. Assume that its base address is in locations BASE through BASE+3 and the index is in locations INDEX through INDEX+3:


```
MOV    EBX,[BASE]           ;GET BASE ADDRESS
MOV    ESI,[INDEX]          ;GET 32-BIT INDEX
MOV    EBX,[EBX+ESI*4]      ;GET 32-BIT ELEMENT
```

Here we use a base register, an index register, and scaled indexing. The displacement is zero.

Jump tables require extra caution. On the 80386, jumps work like other instructions when using based and indexed addressing. That is, the destination is the contents of the effective address, not the effective address itself. The approach here is different from that used on many other processors, such as the Motorola 68000.

Example

Transfer control (jump) to a 32-bit address obtained from a table. Assume that the base address of the table is BASE (a constant) and the index is in locations INDEX through INDEX+3:

```
MOV    ESI,[INDEX]          ;GET INDEX
JMP    BASE[ESI*4]          ;JUMP INDIRECTLY TO
                               ; DESTINATION
```

Be careful here. You might think that the destination is the address $\text{BASE} + \text{ESI} * 4$. It isn't. The destination is the contents of that address. Note, for example, the difference between `JMP EBX`, which jumps to the address in EBX, and `JMP [EBX]`, which jumps to the address at the address in EBX (that is, indirectly through EBX).

The common uses of jump tables are to implement CASE or SWITCH statements (multiway branches in high-level languages such as Ada, C, FORTRAN, Pascal, and PL/I), to decode commands from a keyboard, and to respond to function keys on a terminal. Other uses of jump tables are in selecting I/O drivers, graphics functions, decoding routines, or mathematical methods.

CHARACTER MANIPULATION

The 80386 can manipulate characters as unsigned 8-bit numbers. The letters and digits form ordered subsequences of the ASCII character set.

Examples

1. Jump to address DEST if AL contains ASCII E:

```
CMP AL,'E'           ;IS DATA ASCII E?
JE  DEST             ;YES, BRANCH
```

2. Load register AL with the next character in a string. Assume that the address of the current character is in register ESI. Add 1 to ESI afterward:

```
MOV AL,[ESI]         ;GET NEXT CHARACTER
INC ESI              ;MOVE POINTER
```

3. If register AL contains a nonblank character, store it at the address in EDI and add 1 to EDI afterward:

```
CMP AL,' '           ;IS CHARACTER A BLANK?
JE  NEXTCH
MOV [EDI],AL         ;NO, SAVE IT IN STRING
INC EDI              ;MOVE POINTER
```

NEXTCH: NOP

4. Jump to address DEST if register AL contains a letter between A and F inclusive:

```
CMP AL,'A'           ;IS DATA BELOW A?
JB  DONE             ;YES, DONE
CMP AL,'F'           ;IS DATA A THROUGH F?
JBE DEST             ;YES, BRANCH
```

DONE: NOP ;COME HERE IF NOT A...F

The letters A through F form an ordered subsequence in ASCII (41 through 46 hex). These instructions can validate a hex letter digit.

You can often process strings or arrays faster by using string instructions. As described in Chapter 2, they combine a string operation with the updating of ESI, EDI, or both. The update's sign depends on the D flag (D = 0 for autoincrement or 1 for autodecrement).

The 80386 string primitives are:

CMPS — compare strings. Subtract the operand addressed via EDI from the one addressed via ESI and set the flags accordingly. Neither operand changes.

INS — input string. Load data from an input port (addressed via DX) into the memory location addressed via EDI.

LODS — load string. Load the accumulator from the memory location addressed via ESI.

MOVS — move string. Move data from the memory location addressed via ESI to the one addressed via EDI.

OUTS — output string. Send data from the memory location addressed via ESI to an output port (addressed via DX).

SCAS — scan string. Subtract the contents of the memory location addressed via EDI from the accumulator and set the flags accordingly. Neither operand changes.

STOS — store string. Store the contents of the accumulator at the memory location addressed via EDI.

Note that **CMPS** and **MOVS** update both ESI and EDI. String primitives have byte, word, and double word versions. The versions all update the pointers by the step required to reach the next element (1 for byte versions, 2 for word versions, and 4 for double word versions).

Examples

1. Load AL from the address in ESI, then increase ESI by 1:

```
CLD                ;SELECT AUTOINCREMENTING
LODSB              ;GET DATA AND UPDATE POINTER
```

2. Move a double word from the address in ESI to the one in EDI, then decrease both ESI and EDI by 4:

```
STD                ;SELECT AUTODECREMENTING
MOVSD              ;MOVE A DOUBLE WORD AND UPDATE
                  ; BOTH POINTERS
```

3. Compare the character in AL to the one at the address in EDI. Increase EDI by 1 and jump to DEST if the characters are the same:

```
CLD                ;SELECT AUTOINCREMENTING
SCASB              ;COMPARE CHARACTERS AND UPDATE
                  ; POINTER
JE      DEST       ;JUMP IF THE SAME
```

The **REP** prefix repeats a string instruction while counting down register ECX. As noted in Chapter 2, the sequence of events is:

1. Check if ECX contains zero and exit if it does.
2. Do the string instruction.
3. Subtract 1 from ECX and return to step 1.

Examples

1. Move a block of data from addresses starting at STR1 to ones starting at STR2. The length of the block is in locations LEN through LEN+3:

```
        MOV ESI,STR1      ;BASE ADDRESS OF SOURCE
        MOV EDI,STR2      ;BASE ADDRESS OF DESTINATION
        MOV ECX,[LEN]     ;BLOCK LENGTH
        CLD               ;SELECT AUTOINCREMENTING
    REP  MOVSB             ;MOVE DATA BLOCK
```

MOVSB with the REP prefix forms an implicit loop. It updates the pointers and counters and jumps back automatically. Note that you should consider REP and the subsequent string instruction as a unit for debugging and documentation purposes.

2. Find the next delimiter character (DELIM) in a string starting at the address in EDI. The length of the string is in register ECX. Jump to address DONE if the string has no delimiter:

```
        CLD               ;SELECT AUTOINCREMENTING
        MOV AL,DELIM      ;GET DELIMITER CHARACTER
    REPNE SCASB           ;LOOK FOR DELIMITER CHARACTER
        JNE  DONE         ;JUMP IF NO DELIMITER
```

REP prefixes and string primitives are very useful in parsing command lines. REPNE repeats the string instruction as long as the Zero flag is not 0 and ECX has not been decremented to zero. Note that the processor tests the Zero flag at the end of each iteration, whereas it tests ECX at the beginning.

After the implicit loop, the Zero flag indicates the reason for the exit. $Z = 1$ if it occurred because SCASB found a delimiter, $Z = 0$ if the exit occurred because ECX was decremented to 0. ECX contains the number of characters left in the unparsed part of the string.

CODE CONVERSION

You can convert data between codes using arithmetic or logical operations for simple cases or lookup tables for complex cases.

Examples

1. Convert an ASCII digit in AL to its binary-coded-decimal (BCD) equivalent:

```
SUB AL,'0' ;CONVERT ASCII TO BCD
```

or

```
AND AL,11001111B ;CONVERT ASCII TO BCD
```

The ASCII digits form an ordered sequence (30 to 39 hex). To jump to ERROR if AL is out of range, use

```
SUB AL,'0' ;CONVERT ASCII TO BCD
JC ERROR ;ERROR IF BELOW ASCII ZERO
CMP AL,9
JA ERROR ;ERROR IF ABOVE ASCII NINE
```

2. Convert a decimal digit in AL to its ASCII equivalent:

```
ADD AL,'0' ;CONVERT BCD TO ASCII
```

or

```
OR AL,00110000B ;CONVERT BCD TO ASCII
```

To jump to ERROR if AL is out of range, use

```
CMP AL,9
JA ERROR ;ERROR IF ABOVE NINE
ADD AL,'0' ;CONVERT BCD TO ASCII
```

3. Convert one 8-bit code to another using a lookup table. Assume that the lookup table starts at address NEWCD and the original code is in address CODE:

```
MOV AL,[CODE] ;GET OLD CODE
MOV EBX,NEWCD ;GET BASE ADDRESS OF TABLE
XLAT ;CONVERT TO NEW CODE
```

This approach could convert ASCII to EBCDIC.

MULTIPLE-PRECISION ARITHMETIC

Multiple-precision arithmetic requires a series of operations. For example, multiple-precision addition or subtraction involves the following steps:

1. Clear Carry initially, as there is never a carry into or borrow from the low byte.
2. Add or subtract corresponding units (bytes, words, or double words) using the Add with Carry or Subtract with Borrow instruction.
3. Repeat step 2 until the entire numbers are added or subtracted.

For example, the following program does 128-bit addition as four 32-bit operations:

```
MOV    ECX,4        ;NUMBER OF DOUBLE WORDS = 4
MOV    EDI,NUM1     ;POINT TO LOW BYTES OF NUMBERS
MOV    ESI,NUM2
CLC                                ;CLEAR CARRY INITIALLY
CLD                                ;SELECT AUTOINCREMENTING
ADD32: LODSD         ;GET 32 BITS FROM OPERAND 2
      ADC    EAX,[ECI] ;ADD 32 BITS FROM OPERAND 1
      STOSD        ;STORE RESULT OVER OPERAND 1
      LOOP   ADD32   ;COUNT DOUBLE WORDS
```

Decimal operations must proceed a byte at a time, as packed and unpacked decimal instructions (DAA, DAS, AAA, AAD, AAM, and AAS) operate only on 8-bit results. For example, the following program adds 20-digit packed decimal numbers (2 digits per byte):

```
MOV    ECX,10        ;NUMBER OF BYTES = 10
MOV    EDI,NUM1     ;POINT TO LOW BYTES OF NUMBERS
MOV    ESI,NUM2
CLC                                ;CLEAR CARRY INITIALLY
CLD                                ;SELECT AUTOINCREMENTING
ADDBYT: LODSB        ;GET TWO DIGITS FROM OPERAND 2
      ADC    AL,[EDI] ;ADD TWO DIGITS FROM OPERAND 1
      DAA          ;MAKE SUM DECIMAL
      STOSB        ;STORE SUM OVER OPERAND 1
      LOOP   ADDBYT  ;COUNT BYTES
```

DAA converts a binary sum in AL to a decimal sum.

DATA STRUCTURE MANIPULATION

To handle general data structures (lists, queues, stacks, etc.) on the 80386, we use the procedures described earlier for array manipulation, table lookup, and string processing. The major limitation is the lack of multilevel indirect addressing. However, the LEA (load effective address) instruction can provide this kind of addressing in a series of steps. For more details on data structures, see the books by A. Tenenbaum and M. Augenstein such as *Data Structures Using Pascal*, 2nd ed. (Englewood Cliffs, NJ: Prentice-Hall, 1986).

Examples

1. Get the next element in a linked list. The address of the current element is in register EBX. Each element has a link to its successor in its first 4 bytes. Put the new address in ESI:

```
MOV    ESI,[EBX]           ;REPLACE POINTER WITH LINK
```

You can use a similar procedure to remove an element from a linked list. The only addition is that you must link its predecessor to its successor (thus unlinking it). The sequence is

```
MOV    ESI,[EBX]           ;GET ELEMENT TO BE REMOVED
MOV    EAX,[ESI]           ;MOVE LINK FROM REMOVED
MOV    [EBX],EAX           ; ELEMENT TO ITS PREDECESSOR
```

More generally, if the link is at offset LINK in each element, the sequence is

```
MOV    ESI,LINK[EBX]       ;GET ELEMENT TO BE REMOVED
MOV    EAX,LINK[ESI]       ;MOVE LINK FROM REMOVED
MOV    LINK[EBX],EAX       ; ELEMENT TO ITS PREDECESSOR
```

In practice, you should also test the link to be sure that a successor exists. Remember that MOV does not affect the flags.

2. Insert an element in a linked list. Assume that the element's address is in ESI and the address of the preceding element is in EBX. Link the new element to its predecessor and to its predecessor's successor:

```
MOV    EAX,[EBX]           ;MOVE OLD LINK TO NEW ELEMENT
MOV    [ESI],EAX
MOV    [EBX],ESI           ;MAKE NEW ELEMENT
                               ; INTO NEW LINK
```

This procedure lets you add a new element to a linked list. The new element is now the successor of its predecessor, and the predecessor's former successor is now the new element's successor. More generally, if the links are at offset LINK in each element, the procedure is

```
MOV    EAX,LINK[EBX]       ;MOVE OLD LINK TO NEW ELEMENT
MOV    LINK[ESI],EAX
MOV    LINK[EBX],ESI       ;MAKE NEW ELEMENT
                               ; INTO NEW LINK
```

3. Add an element to a stack. Assume that the address of the next empty stack location is in locations SPTR through SPTR+3 and the new element is in register EAX. Assume also that the stack grows up in memory (toward higher addresses). This is the opposite of the hardware stack, which grows down in memory:


```
MOV    EDI,[SPTR]      ;GET STACK POINTER
CLD                      ;SELECT AUTOINCREMENTING
STOSD                     ;SAVE ELEMENT IN MEMORY
MOV    [SPTR],EDI      ;UPDATE STACK POINTER
```

In practice, we must check for stack overflow by comparing the stack pointer to an upper limit.

4. Remove an element from a stack. Assume that the address of the next empty location is in locations SPTR through SPTR+3. Put the element in register EAX. Assume also that the stack grows up in memory (toward higher addresses):

```
MOV    EDI,[SPTR]      ;GET STACK POINTER
SUB    EDI,4           ;POINT TO HIGHEST OCCUPIED
                        ; LOCATION
MOV    EAX,[EDI]       ;GET ELEMENT FROM STACK
MOV    [SPTR],EDI      ;UPDATE STACK POINTER
```

The lack of predecrementing is a nuisance here. In practice, we must check for stack underflow by comparing the stack pointer to a lower limit.

Program Examples

1. 8-bit maximum value. Assume that register EBX contains the array's base address and register EAX contains its size in bytes. The maximum value ends up in register AL.

```
;
;EXAMINE ELEMENTS ONE AT A TIME, COMPARING EACH
; ONE'S VALUE WITH CURRENT MAXIMUM AND ALWAYS
; KEEPING LARGER VALUE AND ITS ADDRESS.
;IN THE FIRST ITERATION, TAKE THE FIRST ELEMENT
; AS THE CURRENT MAXIMUM
;
```

```
MOV    ECX,EAX         ;SAVE NUMBER OF ELEMENTS
CLD                      ;SELECT AUTOINCREMENTING
MOV    EDI,EBX         ;SET POINTER AS IF PROGRAM HAD
                        ; JUST EXAMINED THE FIRST
                        ; ELEMENT AND FOUND IT TO BE
                        ; LARGER THAN PREVIOUS
                        ; MAXIMUM
```



```

                INC EDI
MAXLP:          MOV EBX,EDI      ;SAVE ADDRESS OF ELEMENT JUST
                                ; EXAMINED AS ADDRESS OF
                                ; MAXIMUM

                DEC EBX
                MOV AL,[EBX]     ;SAVE ELEMENT JUST EXAMINED
                                ; AS MAXIMUM

;
;COMPARE CURRENT ELEMENT TO MAXIMUM
;KEEP LOOKING UNLESS CURRENT ELEMENT IS LARGER
;
MAXLP1:         DEC ECX          ;COUNT ELEMENTS
                JZ   EXITLP      ;JUMP (EXIT) IF ALL ELEMENTS
                                ; EXAMINED

                SCASB            ;COMPARE CURRENT ELEMENT TO
                                ; MAXIMUM, ALSO MOVE POINTER
                                ; TO NEXT ELEMENT

                JAE  MAXLP1       ;CONTINUE UNLESS CURRENT
                                ; ELEMENT IS LARGER

                JB   MAXLP        ;ELSE CHANGE MAXIMUM. TO FIND
                                ; LAST OCCURRENCE RATHER
                                ; THAN FIRST OCCURRENCE,
                                ; CHANGE THE CONDITIONAL
                                ; JUMPS TO JA AND JBE

EXITLP:         NOP             ;EXIT WITH LARGEST ELEMENT IN
                                ; AL AND ITS ADDRESS IN EBX

```

2. Length of a character string. Assume that the base address of the string is in register EDI and the terminating character is in register AL. The length (in register EAX) does not include the terminator:

```

                MOV ECX,-1       ;START STRING LENGTH AT -1
                CLD              ;SET AUTOINCREMENTING
REPNE SCASB     ;KEEP CHECKING CHARACTERS
                                ; UNTIL A TERMINATOR APPEARS.
                                ;CONTINUOUS DECREMENTS OF
                                ; ECX MEAN THAT IT CONTAINS
                                ; -2-LENGTH AT THE END

                MOV EAX,-2       ;COMPUTE STRING LENGTH

```



```
SUB EAX,ECX ; IN REGISTER EAX
```

Typical terminators are an ASCII carriage return and an ASCII NUL (0). A quicker way to compute the string length is with the sequence

```
INC ECX ;COMPUTE STRING LENGTH IN ECX
NOT ECX
```

3. Pattern match. Assume that the base addresses of the strings are in registers ESI and EDI and their length is in register ECX. Set the Zero flag to 1 if the strings match and to 0 if they do not.

```
CLD ;SET AUTOINCREMENTING
REPE CMPSB ;KEEP COMPARING CHARACTERS
; UNTIL ALL ARE EXAMINED OR
; CORRESPONDING CHARACTERS
; ARE NOT THE SAME
```

To set register AL to 1 if the strings match and to 0 if they do not, end the program with

```
SETE AL ;MOVE ZERO FLAG TO AL
```

You could also use SETE to store a boolean value in memory.

4. Convert a hexadecimal digit to ASCII. Assume that the hexadecimal digit is in AL originally:

```
CMP AL,10 ;IS DIGIT 10 OR LARGER?
JB ADDAZ
ADD AL,7 ;YES, ADD AN EXTRA 7
ADDZ: ADD AL,30H ;ADD ASCII ZERO. END WITH ASCII
; DIGIT IN AL
```

The following method, credited to Dennis Allison, works with no branches at all (don't ask me why!):

```
ADD AL,90H ;DEVELOP EXTRA 6 AND CARRY
DAA
ADC AL,40H ;ADD CARRY, ASCII OFFSET
DAA
```

5. Fill a block of memory. Assume that the base address of the block is in register EDI, the value to be stored there is in register AL, and the size of the block is in register ECX:

```
CLD ;SET AUTOINCREMENTING
REP STOSB ;FILL THE BLOCK
```


This will run faster if you fill a word or a double word each time rather than just a byte. Of course, you must divide the size by 2 or 4.

6. Add entry to list. Add the 32-bit element in EAX to a list if it is not already there.

The list starts at address ELEMS and its length is at address COUNT:

```

                MOV EDI,ELEMS           ;GET BASE ADDRESS OF LIST
                MOV ECX,[COUNT]        ;GET LENGTH OF LIST
                CLD                      ;SET AUTOINCREMENTING
REPNE SCASD     ;LOOK FOR ENTRY IN LIST
                JE     DONE             ;EXIT IF ENTRY ALREADY
                                           ; IN LIST
                MOV [EDI],EAX           ;ENTRY NOT IN LIST, SO PUT
                                           ; IT THERE
                INC  DWORD PTR[COUNT];AND ADD 1 TO LENGTH

```

DONE: NOP

At the end of the implied loop (REPNE SCASD), the Z flag is 1 if the exit occurred because the element was found. Z is 0 if the exit occurred because ECX was counted down to zero.

7. Remove entry from list. Remove the 32-bit element in EAX from a list if it is there.

The list starts at address ELEMS and its length is at address COUNT:

```

                MOV EDI,ELEMS           ;GET BASE ADDRESS OF LIST
                MOV ECX,[COUNT]        ;GET LENGTH OF LIST
                CLD                      ;SET AUTOINCREMENTING
REPNE SCASD     ;LOOK FOR ELEMENT IN LIST
                JNE  DONE               ;EXIT IF ELEMENT NOT IN LIST
                MOV ESI,EDI             ;SET POINTERS TO COMPACT
                                           ; REST OF LIST

```

```

                SUB  EDI,4               ; DEST = SOURCE - 4
REP  MOVSD      ;COMPACT REST OF LIST
                DEC  DWORD PTR[COUNT];SUBTRACT 1 FROM LENGTH

```

DONE: NOP

At the end of the implied loop (REPNE SCASD), the Z flag is 1 if the exit occurred because the element was found. Z is 0 if the exit occurred because ECX was counted down to zero.

8. Absolute value. Take the absolute value of the 32-bit number in EAX:.


```
TEST EAX,EAX    ;CHECK SIGN OF NUMBER
JNS  NOTNEG     ;JUMP (DO NOTHING) IF NUMBER
                ; IS POSITIVE
NEG  EAX        ;NEGATE IF NUMBER IS NEGATIVE
NOTNEG: NOP     ;CONTINUE WITH ABSOLUTE VALUE
                ; IN EAX
```

The following method (courtesy of a Microsoft advertisement) uses no branches but changes EDX as well as EAX:

```
CDQ             ;EXTEND HIGH BIT OF NUMBER
                ; INTO EDX, SO [EDX] =
                ; FFFFFFFF IF NUMBER NEGATIVE
                ; AND 0 IF NUMBER POSITIVE
XOR  EAX,EDX     ;TAKE 1'S COMPLEMENT IF
                ; NUMBER NEGATIVE, LEAVE IT
                ; UNCHANGED IF IT IS POSITIVE
                ; XOR WITH 1S INVERTS BITS
SUB  EAX,EDX     ;COMPUTE 2'S COMPLEMENT IF
                ; NUMBER NEGATIVE BY
                ; SUBTRACTING -1. NO EFFECT IF
                ; NUMBER POSITIVE AS [EDX] = 0
```

Remember that EXCLUSIVE-ORing a bit with 1 inverts its value, whereas EXCLUSIVE-ORing it with 0 has no effect.

PARAMETER PASSING TECHNIQUES

The common ways to pass parameters on the 80386 microprocessor are through registers or on the stack. The register approach is fast and simple for subroutines that take only one or two parameters and return only one or two results. The stack approach is more general and can use the special ENTER and LEAVE instructions.

A typical register-based approach extends Intel's standard PL/M-86 function interface as follows:

1. A single data value is passed or returned in AL, AX, or EAX, depending on its length.
2. A single address (offset) is passed or returned in EBX.

This avoids elaborate stack manipulations.

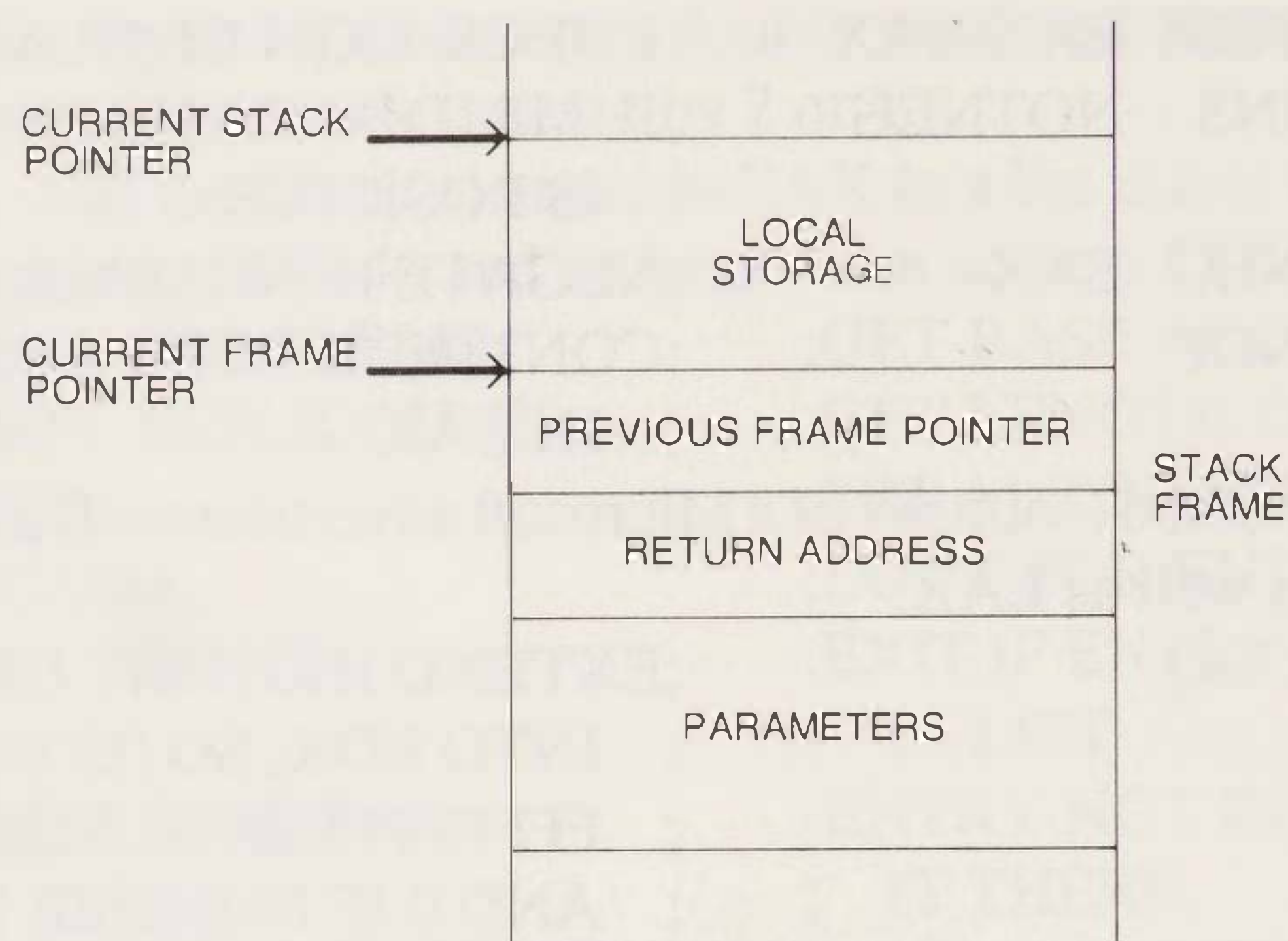


Figure 3-8

Stack contents on entry to the working part of a subroutine.

Stack-based approaches generally use the EBP register. The typical procedure at the start of a subroutine is:

1. Save register EBP in the stack.
2. Set EBP to the current value of the stack pointer (called the *frame pointer*).
3. Reduce the stack pointer by an amount sufficient to allow for local storage within the subroutine.

We refer to the stack area containing parameters, register values, the return address, and local storage as the current *frame*. Figure 3-8 shows a frame with parameters at the bottom (put in the stack by the calling program). Above them are the return address, the previous frame pointer, and local storage. Remember that the stack grows down (toward lower addresses).

The ENTER instruction takes care of all three entry steps. It can even handle nested subroutines that must preserve frame pointers from lower lexical levels. The general form is

```
ENTER      IMM16,IMM8
```

The first operand is the number of bytes in the local storage area. The second operand is the lexical level.

Within a subroutine, you can then address items as follows:

- Parameters with positive offsets from the frame pointer. These offsets must account for the return address and the previous frame pointer.
- Local variables with negative offsets from the frame pointer.

Before returning, the subroutine must clean the stack as follows (see Figure 3-8):

1. Set the stack pointer to the frame pointer, thus removing local variables.
2. Pop the previous frame pointer from the stack.
3. Increase the stack pointer to remove the parameters from the stack.

The LEAVE instruction does the first two steps. A special RET followed by a parameter does the third step. Of course, this approach assumes that the subroutine preserves the frame pointer EBP.

MAKING PROGRAMS RUN FASTER

To speed up a program effectively, you must first find its most frequently executed loops. You must then remove redundant or unnecessary operations from them. Remember that the LEA instruction can help avoid repetitive address calculations.

After this optimization, the best way to speed up 80386 assembly language programs is to eliminate jumps. They are particularly time consuming because they force the processor to clear its pipeline. Thus they cause extra delays while the pipeline is being refilled. Ways to eliminate jumps include:

- Reorganize loops by changing the initial conditions. This often allows a single jump at the end of a loop (a do-until structure rather than a do-while).
- Structure the logic to minimize the number of times a conditional jump is taken. If, for example, one case is infrequent (such as an overflow or an exact match), make it cause a jump rather than having the common case cause one.
- Use in-line code rather than subroutines.

Other ways to reduce execution time include:

- Align all memory accesses. While the 80386 does unaligned accesses, they take extra time. So you should keep procedure entry points, looping destinations, and frequently used data at double-word aligned addresses. You can force alignment by filling data areas, putting NOPs in the program, or using the ALIGN and EVEN assembler directives.

- Use the string primitives and REP whenever possible. They are faster and shorter than sequences with explicit increments and decrements.
- Use registers for their intended purposes. For example, use ECX as a counter, EAX as an accumulator, EBX as a base register, and ESI and EDI as index registers. While the 80386 allows general assignments of registers, there is a time and memory penalty.
- Use other specialized instructions such as XLAT, LOOP, and the BCD and ASCII (packed and unpacked) arithmetic instructions.
- Use instructions that store their results in memory. They can help you avoid saving and restoring registers.

A general way to reduce execution time is by replacing long instruction sequences with tables. A table lookup can replace an instruction sequence if it has no special exits or complex logic. Tables make sense even if many of their entries are the same. After all, it's only cheap memory that you're wasting. What's a mere kilobyte among friends these days? It's not like the long-past 1970s when a man's worth was measured by how many bytes he could remove from his code. Tables take extra memory, but lookup methods are fast, general, easy to program, and easy to change.

COMMON PROGRAMMING ERRORS

The most common errors in 80386 assembly language programs are the following:

- Reversing the order of operands, particularly in MOV and CMP instructions. Remember that the destination comes first. The order is backward in moves but normal in comparisons. That is, CMP op1,op2 computes op1-op2.
- Using the flags incorrectly. The usual problems are trying to jump after instructions (such as MOV or IN) that do not affect the flags and overlooking instructions (such as shifts or SUB reg,reg) that do affect them.
- Getting the logic wrong in conditional jumps. The usual problems are inverting the logic (for example, using JNZ instead of JZ) or jumping incorrectly when operands are equal. Note that comparing equal values clears the Carry. A quick hand check of each jump can avoid many problems.
- Confusing addresses and data. A common typing error is to omit the brackets around an address. A common logical error is to ignore the effects of indirection or to forget how it applies to jumps and calls.

- Mishandling arrays and strings. The usual problem is going beyond the boundaries. Note, for example, that an array starting at BASE and, having N 8-bit elements, occupies addresses BASE through BASE+N-1. The BOUND instruction can help you avoid improper array references.
- Organizing the program improperly. The usual problems are skipping or repeating initialization routines, failing to update counters or pointers, and failing to save results.

SUMMARY

Assembly language programming for the 80386 microprocessor is very similar to programming for its predecessors, the 8086 and 80286. Simple programs can use MOV and arithmetic and logical instructions to manipulate registers. Most instructions have 8-, 16-, and 32-bit versions.

Decision making involves conditional jumps and the bit manipulation and comparison instructions. Bit manipulation instructions move a bit value to the Carry flag. The CMP (compare) instruction affects the Carry, Zero, Sign, and Overflow flags. Conditional jumps are available for all signed and unsigned results.

Multiple-precision arithmetic operations depend on the Carry flag to transfer carries or borrows between iterations. There are also special instructions for packed and unpacked BCD operations. These instructions work on 1 byte at a time and require data to be in the accumulator.

Array, string, table, and data structure manipulation depend on the use of index and base registers. Indexed and based addressing are the key modes, particularly when combined with displacements and scale factors. String primitives simplify string and array handling by combining simple operations with the updating of one or two pointers. The REP prefix forms an implicit loop that repeats a string primitive a specific number of times. The LOOP instruction provides a simple decrement-and-jump-if-not-zero capability.

After standard optimization techniques have been used, the best way to make 80386 programs run faster is by reducing the number of jumps. The most common errors in 80386 programs are reversing the order of operands, using the flags incorrectly, inverting decision logic, confusing addresses and data, handling arrays and strings incorrectly, and organizing sequences improperly.

Input/Output

*It (marriage) happens as with cages: the birds without
despair to get in, and those within despair to get out.*

Michel Eyquem de Montaigne, *Essays*

*Talk of court news; and we'll talk with them too,
Who loses and who wins; who's in, who's out;
And take upon's the mystery of things. . . .*

William Shakespeare, *King Lear*

This chapter describes 80386 input/output. It first explains alternative I/O methods. It then discusses addressing, I/O instructions, and I/O chips. Later sections present examples and describe interrupts and direct memory access (DMA).

ALTERNATIVE I/O METHODS

The major approaches to I/O with any processor are the following:

1. Programmed I/O. Here everything occurs under program control. Software must determine whether devices are ready and select the order in which to handle them.

We refer to checking a device's status as *polling*. This method is best suited to low-speed peripherals such as switches and small displays.

2. Interrupt-driven I/O. Here a special signal that goes directly into the CPU (rather than to an I/O port) indicates that a device is requesting a transfer. The processor responds by suspending its current activities and servicing the request. This method is best suited to medium-speed peripherals such as keyboards, terminals, communications lines, plotters, and printers.
3. DMA (direct memory access) transfers. Here data moves directly between memory and I/O without processor intervention. The CPU must usually load external registers and counters with a base address and the block size. The processor then simply waits in a suspended state while the actual transfer occurs. This method is best suited to high-speed peripherals such as disks and signal processing or image processing boards.

The 80386 allows all these methods. It also provides a clocked approach called *block input/output* that falls between interrupt-driven I/O and DMA in performance. This method transfers a fixed amount of data between memory and a peripheral that runs at or close to processor speed.

I/O ADDRESSING

The 80386 allows two ways of addressing I/O ports:

- Separate I/O addresses (called *isolated input/output*)
- Allocation of memory addresses to input/output (called *memory-mapped input/output*)

Figures 4-1 and 4-2 show how these approaches assign space to I/O and memory addresses.

Isolated I/O is by far the more common approach. The 80386 allows 64K of separate I/O addresses. The actual ports may be any combination of 8-, 16-, and 32-bit units. I/O instructions use only the 16 least significant address bits. There is no distinction between logical and physical addresses, and neither segmentation nor paging applies.

Isolated I/O clearly separates the I/O and memory sections. One reason for doing this is that the sections have different basic units. Memory comes in units of thousands or millions of bytes. Input/output, on the other hand, comes in the form of individual ports or packages containing a few ports. Handling both sections with one addressing

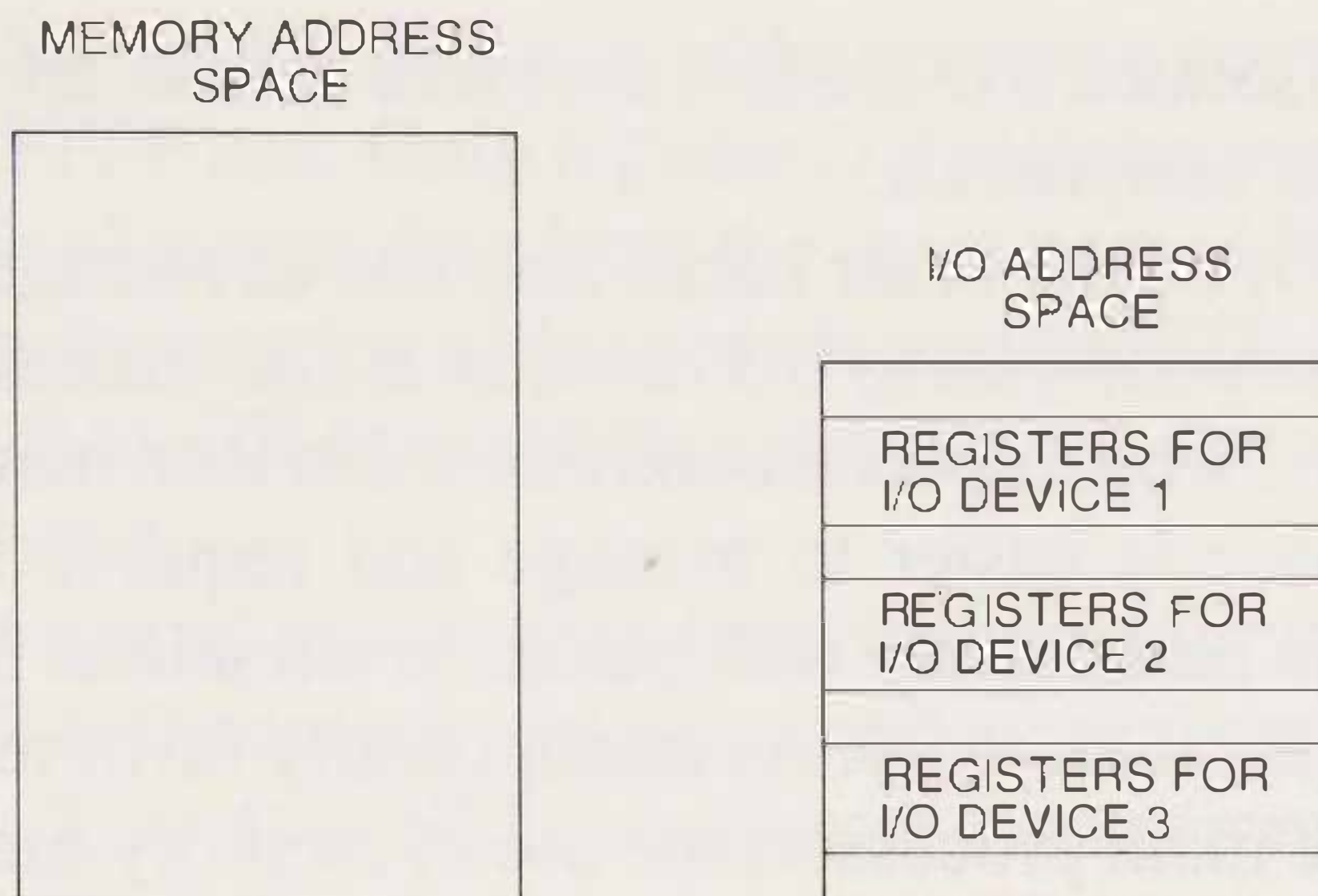


Figure 4-1

Isolated I/O with separate memory and I/O address spaces.

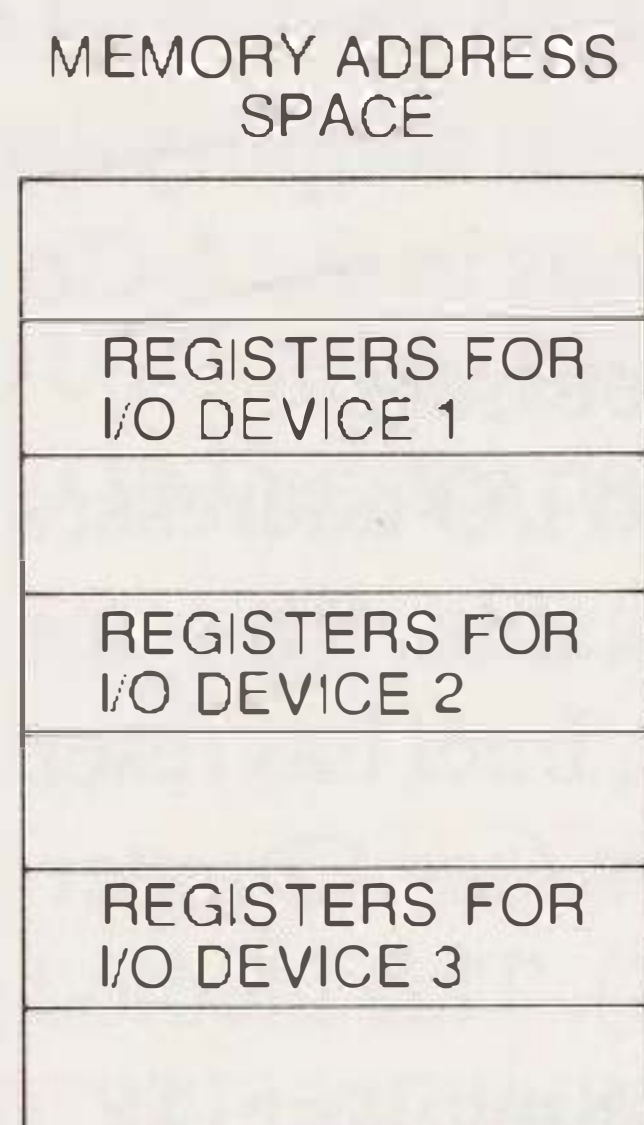


Figure 4-2

Memory-mapped I/O with a single address space.

system is like assigning space on a city block to both huge skyscrapers and single-family homes.

Isolated I/O also makes it easy to find I/O instructions in programs for debugging or analysis. Besides, it emphasizes the fact that I/O and memory behave differently. Peripherals are generally unidirectional, usually operate much more slowly than the CPU, and may not retain their contents like a memory. For example, no matter how one addresses a keyboard or printer, neither behaves like a memory. The keyboard does not respond to data sent to it, nor does the printer have much of interest to say. Neither peripheral can operate nearly as fast as the processor. Furthermore, keyboard data changes as the operator presses keys, and output may not be readable even while the printer

is working on it. Thus isolated I/O is often the more natural approach, particularly for low- and medium-speed peripherals.

Memory-mapped I/O, on the other hand, also has advantages. It lets the programmer apply the entire instruction set to I/O devices. It also makes it easy to direct I/O to or from memory buffers, where special controllers can then handle it at their own rate. The single address space is easier to manage and requires fewer control signals. Memory-mapped I/O is particularly well-suited to situations involving complex I/O devices with their own local memory. As trends clearly favor removing major responsibility for I/O from the main processor, the use of memory-mapped I/O will probably increase in the future.

Note that memory-mapped I/O reduces a computer's memory capacity. The system must dedicate some address space to I/O, as shown in Figure 4-2. Because of hardware constraints and fixed assignments, this space is in the middle in most PCs. The result is a discontinuous address space and a great deal of confusion. In practice, the I/O address space is often made quite large to simplify decoding. However, wasting thousands of addresses is clearly far less serious in the 4-Gb space of the 80386 than it was in the more limited spaces of earlier processors.

Instructions can specify isolated I/O addresses in two ways on the 80386:

- As immediate 8-bit constants. This method applies only to ports 00 through FF hex. However, Intel has reserved ports F8 through FF for use with numeric coprocessors (see Chapter 8).
- Via register DX (*not* EDX). This method provides access to any I/O port but at the cost of reserving register DX. The dedication of DX conflicts with its other uses, such as in extension, division, and multiplication. No alternative to DX is allowed.

I/O INSTRUCTIONS

The 80386's I/O instructions are:

- IN accumulator, port address and OUT port address, accumulator moves data between an accumulator and the specified absolute port address. The accumulator may be AL, AX, or EAX. The port address must be in the range 00 through FF (actually 00 through F7 as Intel reserves F8 through FF for the coprocessor).
- IN accumulator, DX and OUT DX, accumulator moves data between an accumulator and the port addressed via register DX. The accumulator

may be AL, AX, or EAX. The port address may be anywhere in the range 0000 through FFFF hex. Only register DX may be used in this way.

- INS (input string) moves data from the input port addressed via register DX to the memory location addressed via register EDI. It then either adds a step (1, 2, or 4) to EDI or subtracts a step from it, depending on whether the D flag is 0 or 1.
- OUTS (output string) moves data from the memory location addressed via register ESI to the output port addressed via register DX. It then either adds a step (1, 2, or 4) to ESI or subtracts a step from it, depending on whether the D flag is 0 or 1.

Both INS and OUTS have byte, word, and double word versions. As with other string instructions, the step size depends on the amount of data transferred. Note that INS uses register EDI as its pointer, whereas OUTS uses ESI.

In INS and OUTS, the data never passes through a user register. The result is like a low-speed DMA channel operating under program control. In particular, note that the accumulator is not involved.

We can repeat either INS or OUTS with REP, producing block input/output. However, the peripheral must be able to transfer data at close to processor speed. The tight loops

REP INS

or

REP OUTS

do not allow a software delay between operations. The peripheral must be able to provide or accept new data each time it is addressed. The hardware can impose a short delay. In practice, this approach makes sense only for special controllers. Of course, INS and OUTS may still be useful steps in longer loops without REP.

PROGRAMMABLE I/O CHIPS

Most 80386 I/O sections contain programmable I/O chips. These devices have many different operating modes. A program selects among them by storing values in control or command registers. The advantages of programmable I/O devices are:

- They take less board space, use less power, and are cheaper to install than circuits made from less integrated parts. These are obviously important factors in personal computers.

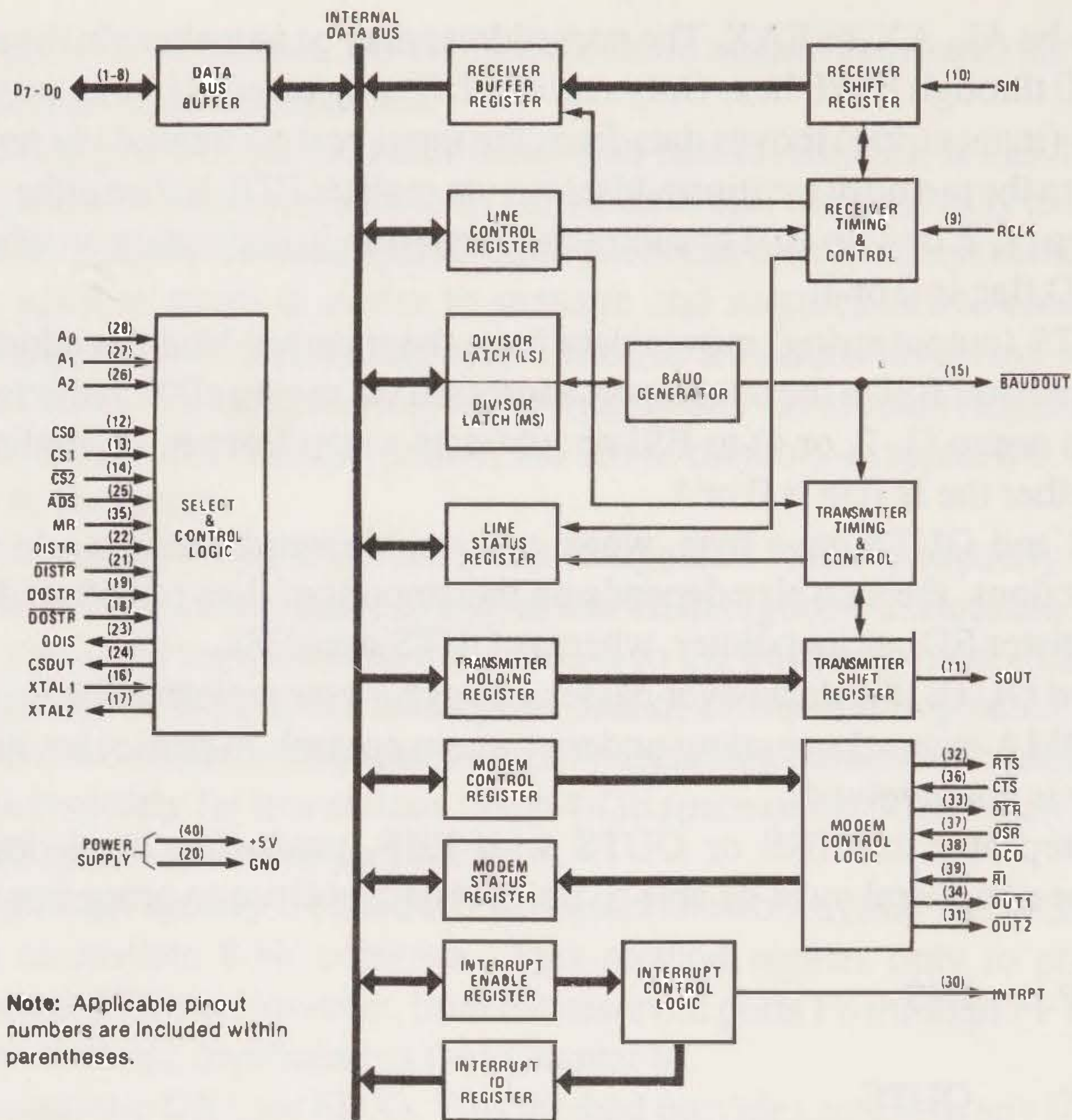


Figure 4-3

Block diagram of the 8250 Asynchronous Communications Element (ACE).

- They can handle a wide range of applications, so boards and computers based on them can serve many purposes. This is clearly an advantage to manufacturers of chips, boards, and computers.
 - Changes and corrections can be made in software rather than in hardware.
- Programmable I/O devices also have some disadvantages such as:
- Lack of standardization. Each part has its own set of operating modes and ways to select them. The user must generally learn the idiosyncrasies

of a particular chip from experience. For most users, the chips are “black boxes” that do their jobs in some unknown way.

- Higher cost than less integrated parts.
- Limited documentation. The user must rely on manufacturers’ data sheets and an occasional application note.

The following programmable I/O devices are popular in 80386-based computers:

- 8250 asynchronous communications element (ACE), a serial (1-bit) interface. Lucky they didn’t call it the asynchronous serial signaler. We often call devices like the 8250 UARTs (*universal asynchronous receiver/transmitters*).
- 8255 programmable peripheral interface (PPI), a parallel (8- or 16-bit) interface. Don’t ask me what the name means. It’s probably computerese for “thingamajig.”
- 8253 or 8254 programmable interval timer (PIT), a timing device.

Many similar devices are available from several manufacturers. We will describe the 8250, 8255, and 8253 briefly only because of their widespread use in computers such as IBM PCs. They are also simpler and easier to understand than many newer devices, yet they do the same functions.

8250 ACE

The 8250 ACE is a popular interface between a microprocessor that handles data in parallel (8, 16, or 32 bits at a time) and peripherals that handle data serially (1 bit at a time). Serial interfaces are popular (particularly over long distances) because of their low cost. After all, they require only one data line. Common serial peripherals include terminals, modems, printers, and mice (mouses?). Figure 4-3 is a block diagram of the 8250 device. Its features include:

- Interrupt-driven or programmed operation.
- Buffering to eliminate the need for precise synchronization. The device thus holds the data until the processor or peripheral is ready for it.
- Modem and RS-232 control signals. RS-232 is a standard serial interface defined by the Electronic Industries Association (EIA).
- Choice of common options for number of bits per character (5 to 8), parity (even, odd, or none), and number of stop bits (idle periods) between characters (1, 1 1/2, or 2).

Table 4-1
Summary of 8250 ACE registers

Bit No.	Register Address										
	0 DLAB = 0	0 DLAB = 0	1 DLAB = 0	2	3	4	5	6	7	0 DLAB = 1	1 DLAB = 1
	Receiver Buffer Register (Read Only)	Transmitter Holding Register (Write Only)	Interrupt Enable Register	Interrupt Ident. Register (Read Only)	Line Control Register	MODEM Control Register	Line Status Register	MODEM Status Register	Scratch Register	Divisor Latch (LS)	Latch (MS)
	RBR	THR	IER	IIR	LCR	MCR	LSR	MSR	SCR	DLL	DLM
0	Data Bit 0*	Data Bit 0	Enable Received Data Available Interrupt (ERBFI)	"0" if Interrupt Pending	Word Length Select Bit 0 (WLS0)	Data Terminal Ready (DTR)	Data Ready (DR)	Delta Clear to Send (DCTS)	Bit 0	Bit 0	Bit 8
1	Data Bit 1	Data Bit 1	Enable Transmitter Holding Register Empty Interrupt (ETBEI)	Interrupt ID Bit (0)	Word Length Select Bit 1 (WLS1)	Request to Send (RTS)	Overrun Error (OE)	Delta Data Set Ready (DDSR)	Bit 1	Bit 1	Bit 9
2	Data Bit 2	Data Bit 2	Enable Receiver Line Status Interrupt (ELSI)	Interrupt ID Bit (1)	Number of Stop Bits (STB)	Out 1	Parity Error (PE)	Trailing Edge Ring Indicator (TERI)	Bit 2	Bit 2	Bit 10
3	Data Bit 3	Data Bit 3	Enable MODEM Status Interrupt (EDSSI)	0	Parity Enable (PEN)	Out 2	Framing Error (FE)	Delta Data Carrier Detect (DDCD)	Bit 3	Bit 3	Bit 11
4	Data Bit 4	Data Bit 4	0	0	Even Parity Select (EPS)	Loop	Break Interrupt (BI)	Clear to Send (CTS)	Bit 4	Bit 4	Bit 12
5	Data Bit 5	Data Bit 5	0	0	Stick Parity	0	Transmitter Holding Register (THRE)	Data Set Ready (DSR)	Bit 5	Bit 5	Bit 13
6	Data Bit 6	Data Bit 6	0	0	Set Break	0	Transmitter Empty (TEMT)	Ring Indicator (RI)	Bit 6	Bit 6	Bit 14
7	Data Bit 7	Data Bit 7	0	0	Divisor Latch Access Bit (DLAB)	0	0	Data Carrier Detect (DCD)	Bit 7	Bit 7	Bit 15

*Bit 0 is the least significant bit. It is the first bit serially transmitted or received.

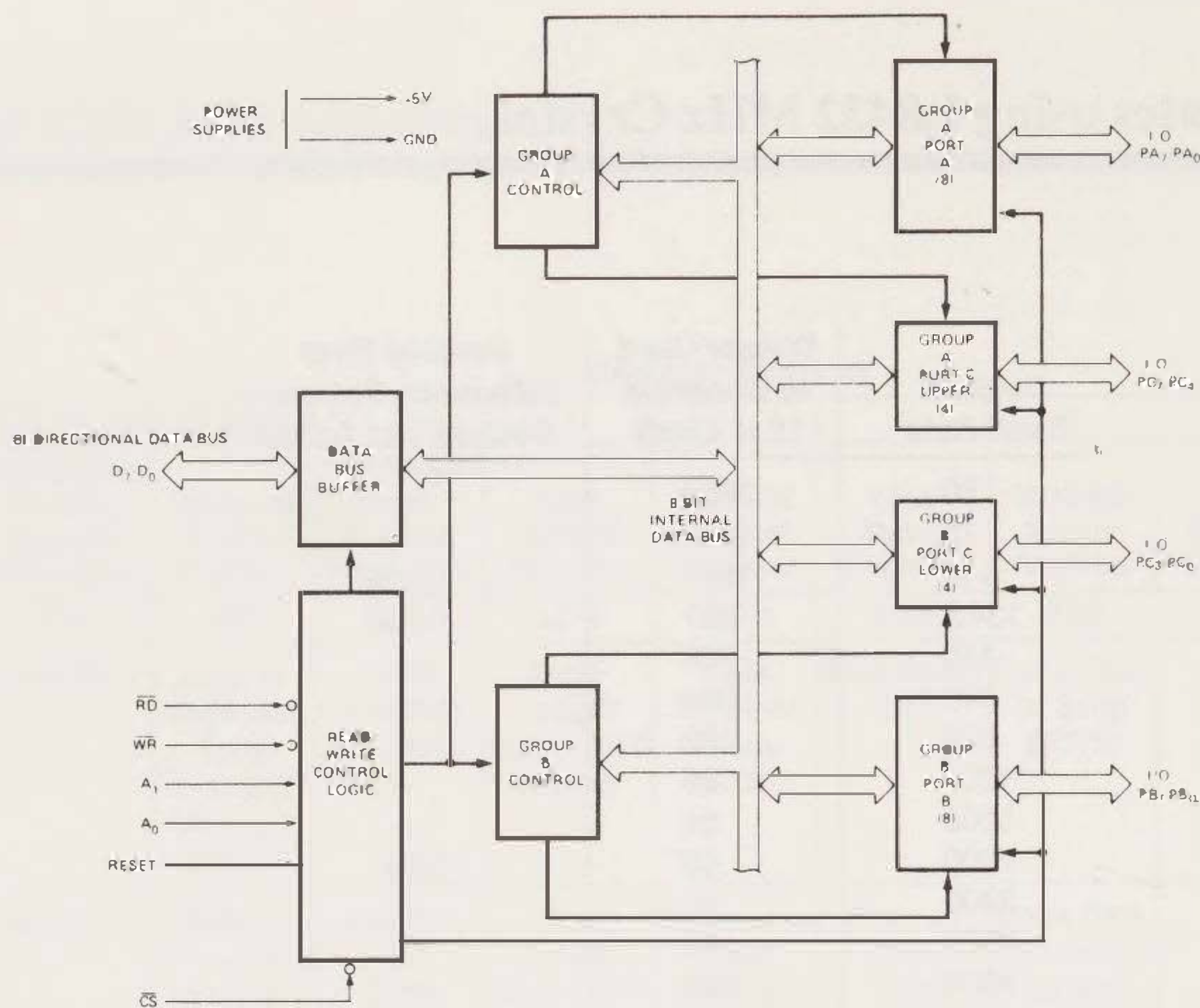
Table 4-2
8250 ACE baud rates using 1.8432 MHz Crystal

Desired Baud Rate	Divisor Used to Generate 16 × Clock	Percent Error Difference Between Desired and Actual
50	2304	—
75	1536	—
110	1047	0.026
134.5	857	0.058
150	768	—
300	384	—
600	192	—
1200	96	—
1800	64	—
2000	58	0.69
2400	48	—
3600	32	—
4800	24	—
7200	16	—
9600	12	—
19200	6	—
38400	3	—
56000	2	2.86

Note: 1.8432 MHz is the standard 8080 frequency divided by 10.

- Timer (called a *baud rate generator*) that provides the intervals between bits.
- Error-checking facilities. These include the ability to screen out short noise pulses and to recognize problems such as overrun, incorrect parity, and improper data structure (or *framing*). Overrun means that the device received a new character before the old one was read.

Table 4-1 summarizes the 8250's registers. Data passes through register 0. The others act as control or status registers. The baud rate generator can produce any common data rate (such as 110, 300, 1200, 2400, 4800, or 9600 baud) with the aid of a divisor as shown in Table 4-2.

**Figure 4-4**

Block diagram of the 8255 Programmable Peripheral Interface (PPI).

8255 PPI

The 8255 PPI is a parallel interface intended for peripherals such as printers, plotters, and analog I/O boards that handle data 8 or 16 bits at a time. Figure 4-4 is a block diagram of the device. Its key features are:

- Two 8-bit ports and two 4-bit ports that can be either input or output
- Direct bit set/reset capability for the status and control port (port C)
- Automatic status and control signals for either unidirectional or bidirectional transfers

Figures 4-5 and 4-6 summarize the PPI's options. Its operating modes are:

- Mode 0, in which all ports work independently as either inputs or outputs. There are separate input or output selection bits for the top and bottom halves of port C (bits 0 through 3 and 4 through 7).
- Mode 1, in which bits of port C act as status and control signals for ports A and B. The status signals (STROBE or ACKNOWLEDGE) indicate whether the peripheral is ready to transfer data. The control signals (IN-

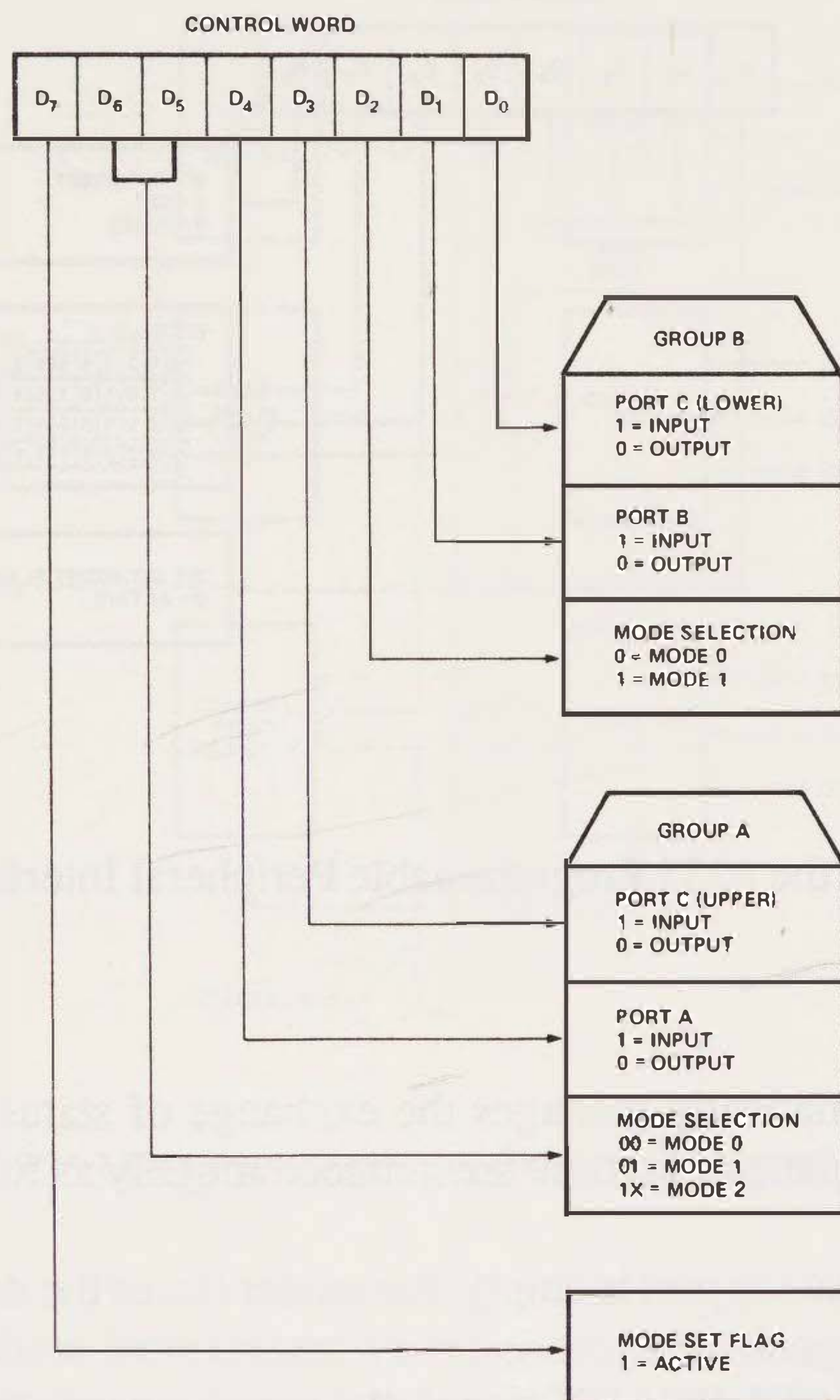


Figure 4-5

Mode definition format for the 8255 Programmable Peripheral Interface (PPI).

TERRUPT REQUEST and BUFFER FULL) request service from the processor and tell the peripheral whether a transfer may proceed.

- Mode 2, in which most of port C acts as status and control signals for bidirectional transfers through port A. This mode is uncommon in practice.

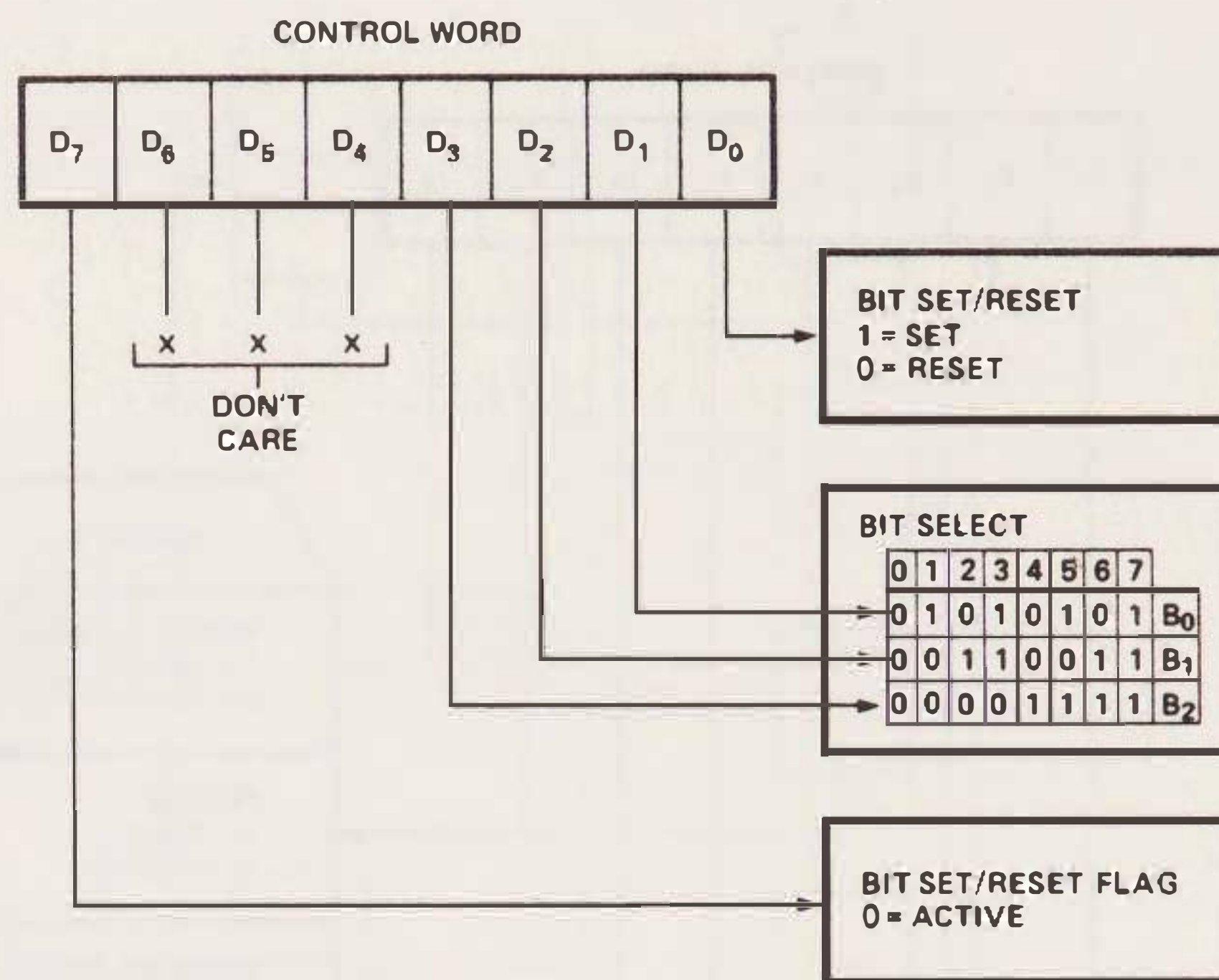


Figure 4-6

Bit set/reset format for the 8255 Programmable Peripheral Interface (PPI).

The 8255 PPI automatically manages the exchange of status and control signals (called *handshaking*). Handshake transfers proceed roughly as follows:

1. After determining that the port is empty, the sender stores the data in it and activates a DATA READY signal.
2. In response to the DATA READY signal, the receiver reads the data from the port and activates a DATA ACCEPTED signal.

The signals thus validate the transfer, much as a human handshake signifies the acceptance of an agreement. The 8255 PPI not only generates and latches (holds) signals but it also activates and deactivates them without further processor intervention.

8253 and 8254 PITs

The 8253 and 8254 timers are often used to generate real-time clocks and to measure time intervals between events. Figure 4-7 is a block diagram of the 8253 device.

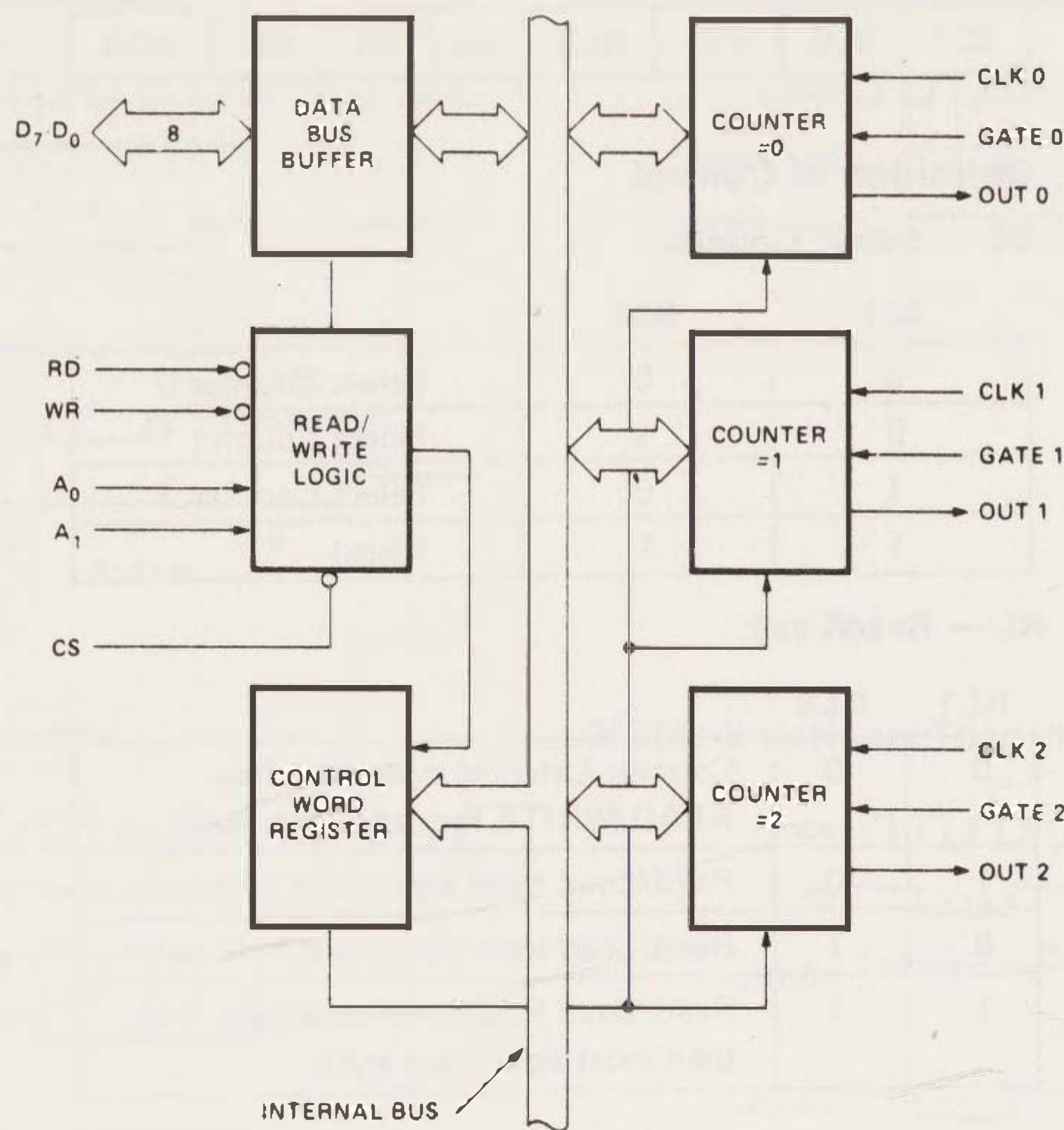


Figure 4-7

Block diagram of the 8253 Programmable Interval Timer (PIT).

Both timers have three independent 16-bit counters and can count in either binary or decimal. Their operating modes (selected with control words as shown in Figure 4-8) can create the following waveforms (see Figure 4-9):

- Single wide pulse
- Periodic series of narrow pulses
- Square wave
- Single narrow pulse at the end of an interval

I/O EXAMPLES

1. Jump to location SERVSW if a switch attached to bit BITNO of port IPORT is closed (0):

```
MOV DX,IPORT    ;ACCESS INPUT PORT
```


Control Word Format

D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
SC1	SC0	RL1	RL0	M2	M1	M0	BCD

Definition of Control
SC — Select Counter:

SC1	SC0	
0	0	Select Counter 0
0	1	Select Counter 1
1	0	Select Counter 2
1	1	Illegal

RL — Read/Load:

RL1	RL0	
0	0	Counter Latching operation (see READ/WRITE Procedure Section)
1	0	Read/Load most significant byte only.
0	1	Read/Load least significant byte only.
1	1	Read/Load least significant byte first, then most significant byte.

M — MODE:

M2	M1	M0	
0	0	0	Mode 0
0	0	1	Mode 1
X	1	0	Mode 2
X	1	1	Mode 3
1	0	0	Mode 4
1	0	1	Mode 5

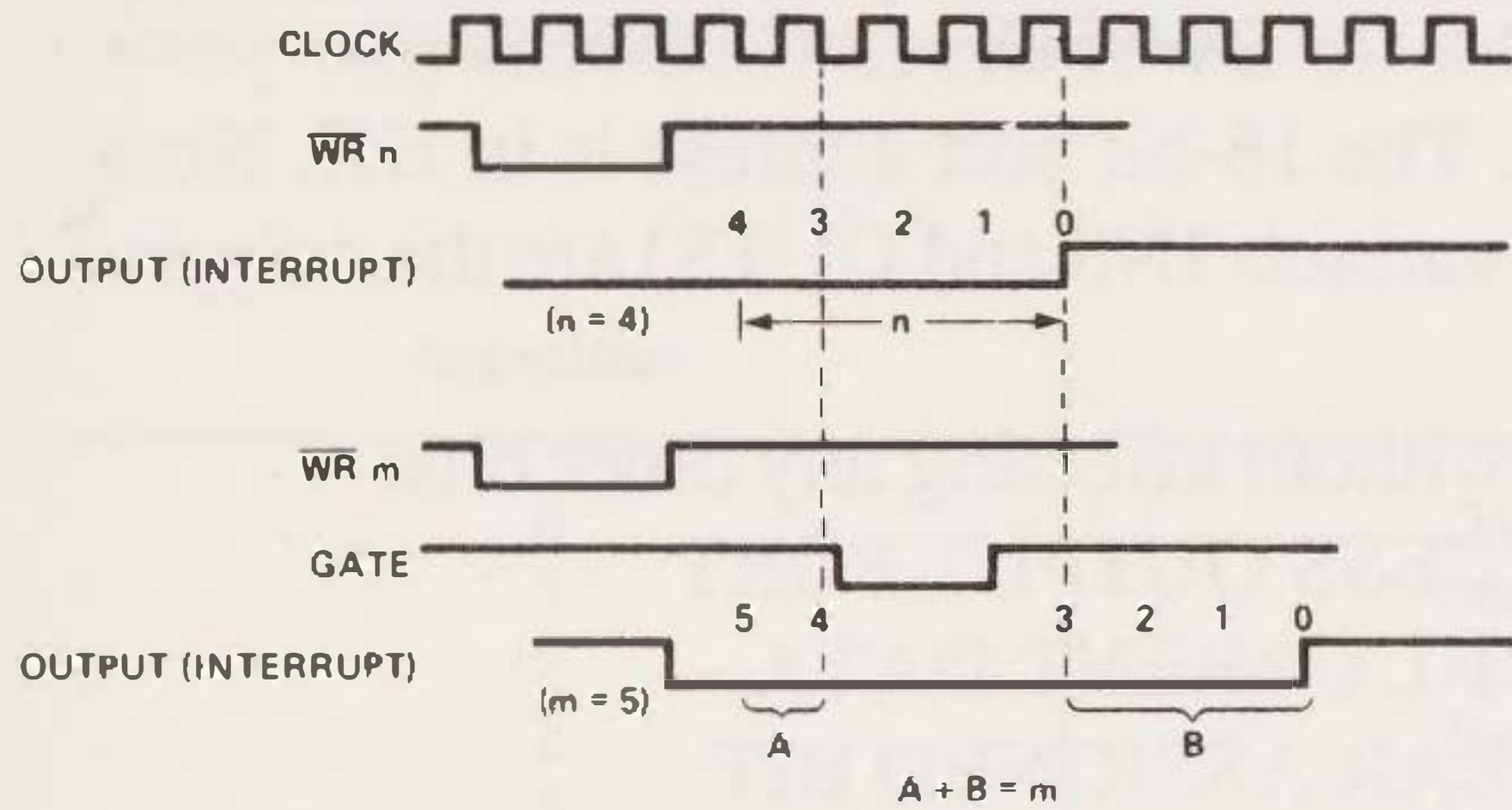
BCD:

0	Binary Counter 16-bits
1	Binary Coded Decimal (BCD) Counter (4 Decades)

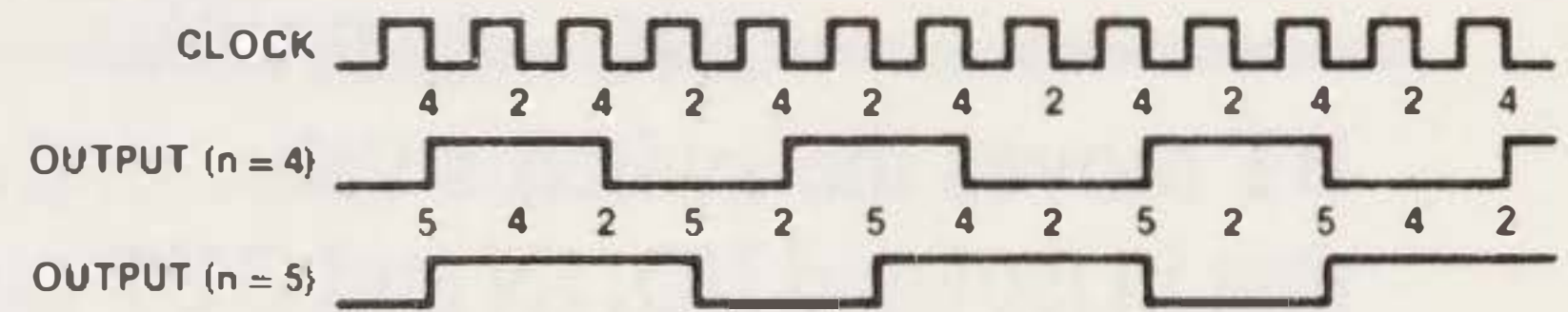
Figure 4-8

Control word format for the 8253 Programmable Interval Timer (PIT).

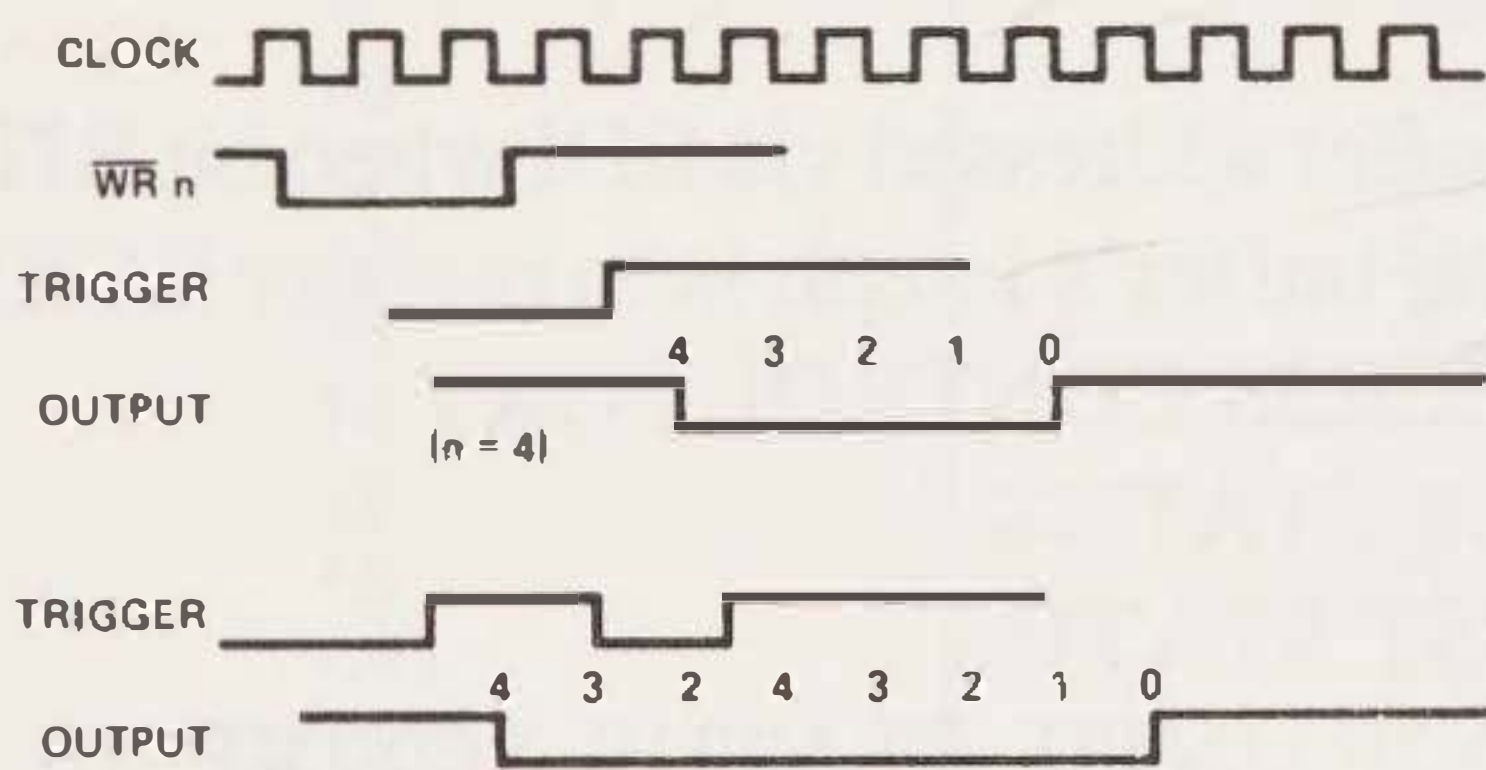
MODE 0: Interrupt on Terminal Count



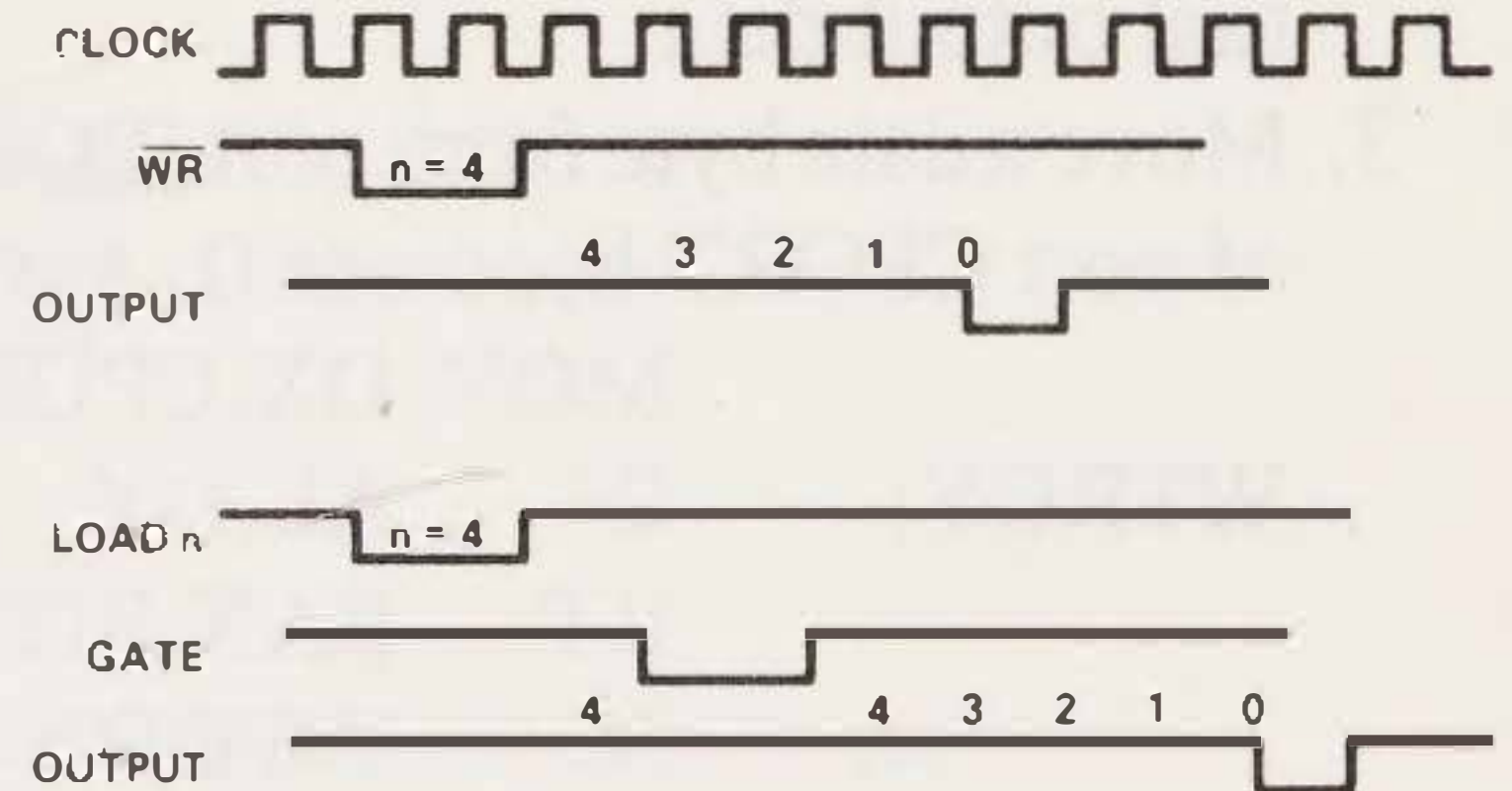
MODE 3: Square Wave Generator



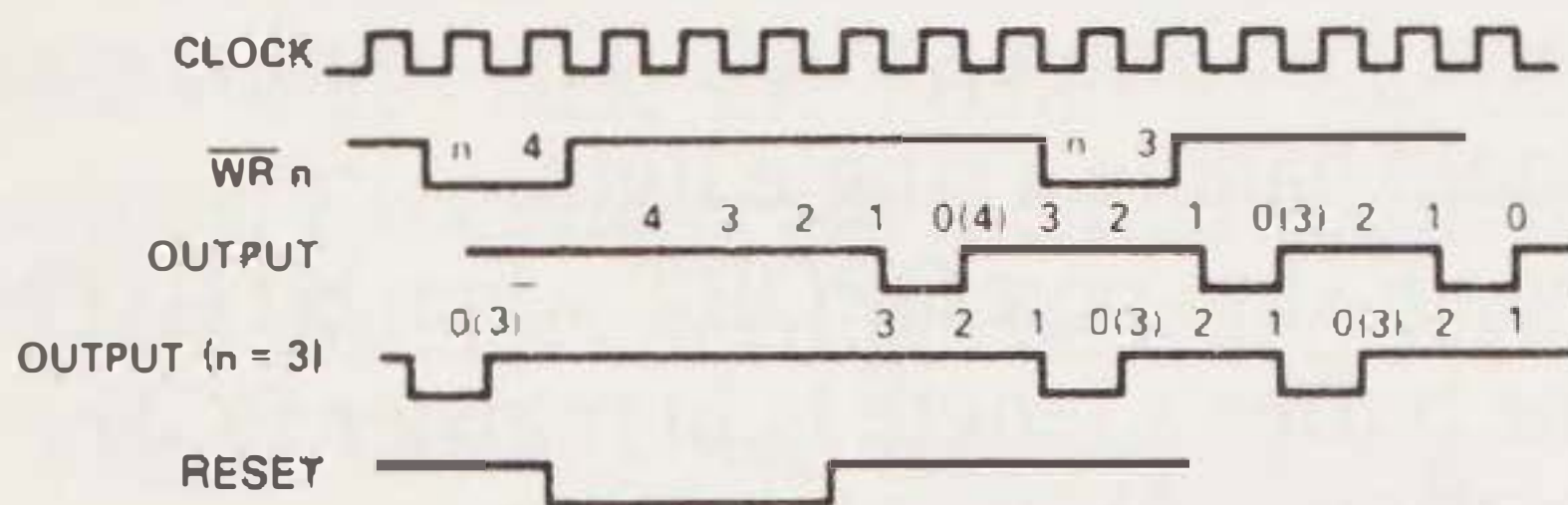
MODE 1: Programmable One-Shot



MODE 4: Software Triggered Strobe



MODE 2: Rate Generator



MODE 5: Hardware Triggered Strobe

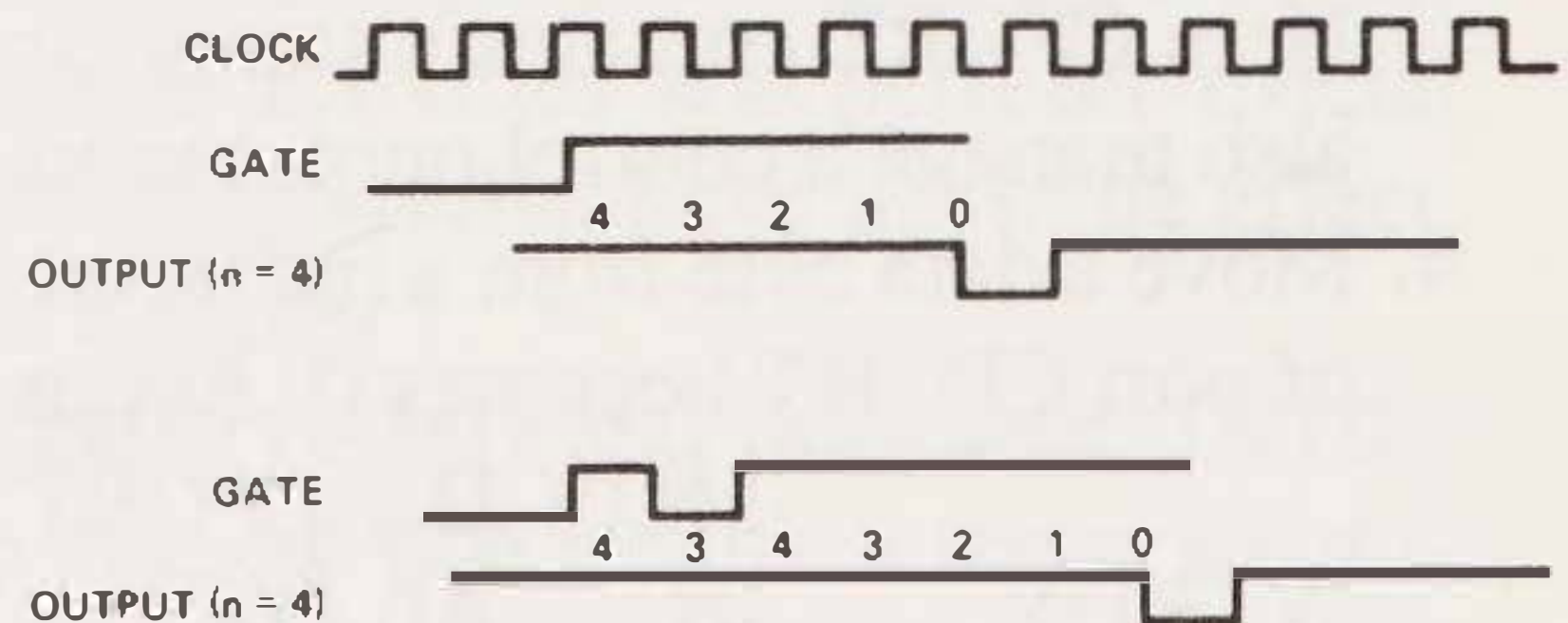


Figure 4-9

Timing diagrams for operating modes of 8253 Programmable Interval Timer (PIT).


```

IN    AL,DX      ;GET DATA
BT    EAX,BITNO  ;TEST SWITCH
JNC   SERVSW     ;JUMP IF SWITCH IS CLOSED (0)

```

BT moves the switch's value to the Carry. The 16-bit port address is in DX. Note that in isolated I/O, IN and OUT (and their variants INS and OUTS) are the only instructions that apply to ports.

2. Clear bit position BITNO of port OPORT without affecting any other bits:

```

MOV DX,OPORT     ;ACCESS OUTPUT PORT
IN    AL,DX      ;GET CURRENT DATA
BTR   EAX,BITNO  ;CLEAR ASSIGNED BIT
OUT   DX,AL      ;SEND DATA WITH BIT CLEARED

```

This routine assumes that the output port is readable. If it is not, you must save a copy of the data in memory. The program must update the copy as well as the actual output data.

3. Move a data byte from port IPORT into a buffer addressed via EDI when bit BITNO of port CPORT becomes 0. Assume that the buffer's length is in register ECX:

```

WTRDY:  MOV DX,CPORT    ;ACCESS CONTROL PORT
        IN    AL,DX     ;GET STATUS
        BT    EAX,BITNO ;TEST STATUS
        JC    WTRDY     ;LOOP UNTIL STATUS ACTIVE (0)
        CLD           ;SELECT AUTOINCREMENTING
        MOV DX,IPORT    ;ACCESS INPUT PORT
        INSB          ;MOVE DATA TO BUFFER
        INC   ECX       ;ADD 1 TO BUFFER LENGTH

```

Here CPORT contains a status input that acts just like a switch. The routine could also manage a control output much as it would handle a single light.

4. Move a data byte from a buffer addressed via ESI to port OPORT when bit BITNO of port CPORT becomes 0. Assume that the buffer's length is in register ECX:

```

WTRDY:  MOV DX,CPORT    ;ACCESS CONTROL PORT
        IN    AL,DX     ;GET STATUS
        BT    EAX,BITNO ;TEST STATUS
        JC    WTRDY     ;LOOP UNTIL STATUS ACTIVE (0)
        CLD           ;SELECT AUTOINCREMENTING
        MOV DX,OPORT    ;ACCESS OUTPUT PORT
        OUTSB          ;SEND DATA FROM BUFFER
        INC   ECX       ;ADD 1 TO BUFFER LENGTH

```

The contents of the control port are still in AL at the end of the routine.

Table 4-3
80386 interrupts and exceptions in numerical order

Identifier	Description
0	Divide error
1	Debug exceptions
2	Nonmaskable interrupt
3	Breakpoint (one-byte INT 3 instruction)
4	Overflow (INTO instruction)
5	Bounds check (BOUND instruction)
6	Invalid opcode
7	Coprocessor not available
8	Double fault
9	(reserved)
10	Invalid TSS
11	Segment not present
12	Stack exception
13	General protection
14	Page fault
15	(reserved)
16	Coprocessor error
17-31	(reserved)
32-255	Available for external interrupts via INTR pin

INTERRUPTS

More advanced I/O methods require a discussion of interrupts. *Interrupts* are external signals that cause the 80386 to suspend its normal activities and perform special routines. *Exceptions* are internal conditions or instructions that have the same effect. We will discuss interrupts here and exceptions in Chapter 7.

The 80386 has two interrupt inputs: nonmaskable (NMI) and maskable (INTR). Nonmaskable means that the interrupt cannot be shut out (the technical terms are *disarmed*, *masked*, or *disabled*). The maskable interrupt is controlled by IF (the interrupt enable flag). IF is bit 9 of the flags register (see Figure 2-3). Setting IF enables maskable interrupts, clearing it disables them.

Each interrupt has an identifier or *interrupt type* (see Table 4-3). So do exceptions, which include conditions such as divide errors, invalid operation codes, and other problems that we have not yet described. Note that Intel reserves interrupt types 0 through 31 for its own purposes, although it has not yet specified the meanings of all

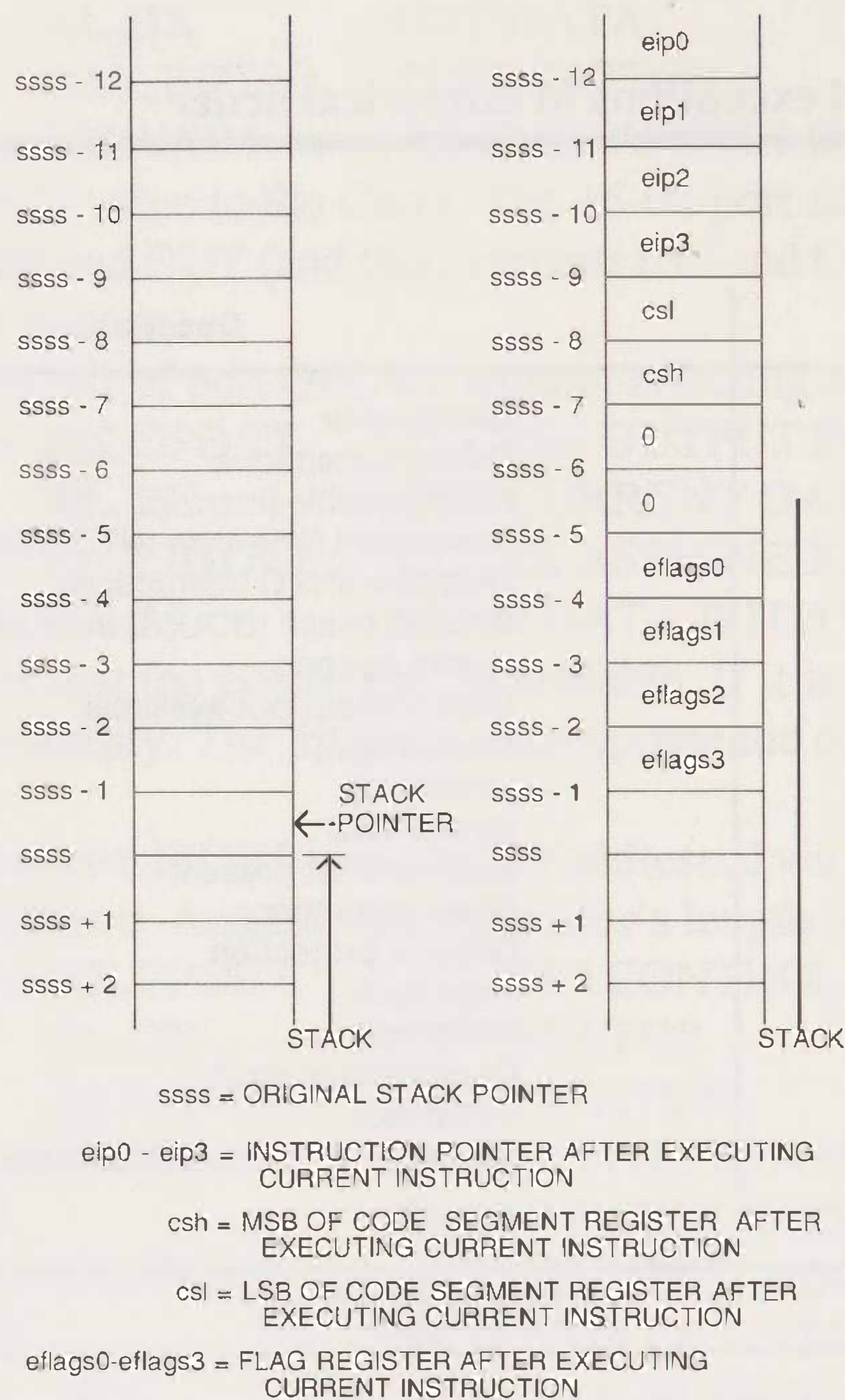


Figure 4-10

Saving the status of the 80386 microprocessor in the stack.

of them. In the past, however, Microsoft and IBM have not respected Intel's claims and have used reserved interrupts for MS-DOS functions.

The 80386 responds to an interrupt or exception as follows:

1. It saves the current flags, instruction pointer, and code segment register in the stack. Figure 4-10 shows the order, assuming 32-bit registers (CS is extended with zeros). The 16-bit case is similar, except that CS is not extended.
2. It sets the IF so that other maskable interrupts will not be recognized until they are specifically allowed.

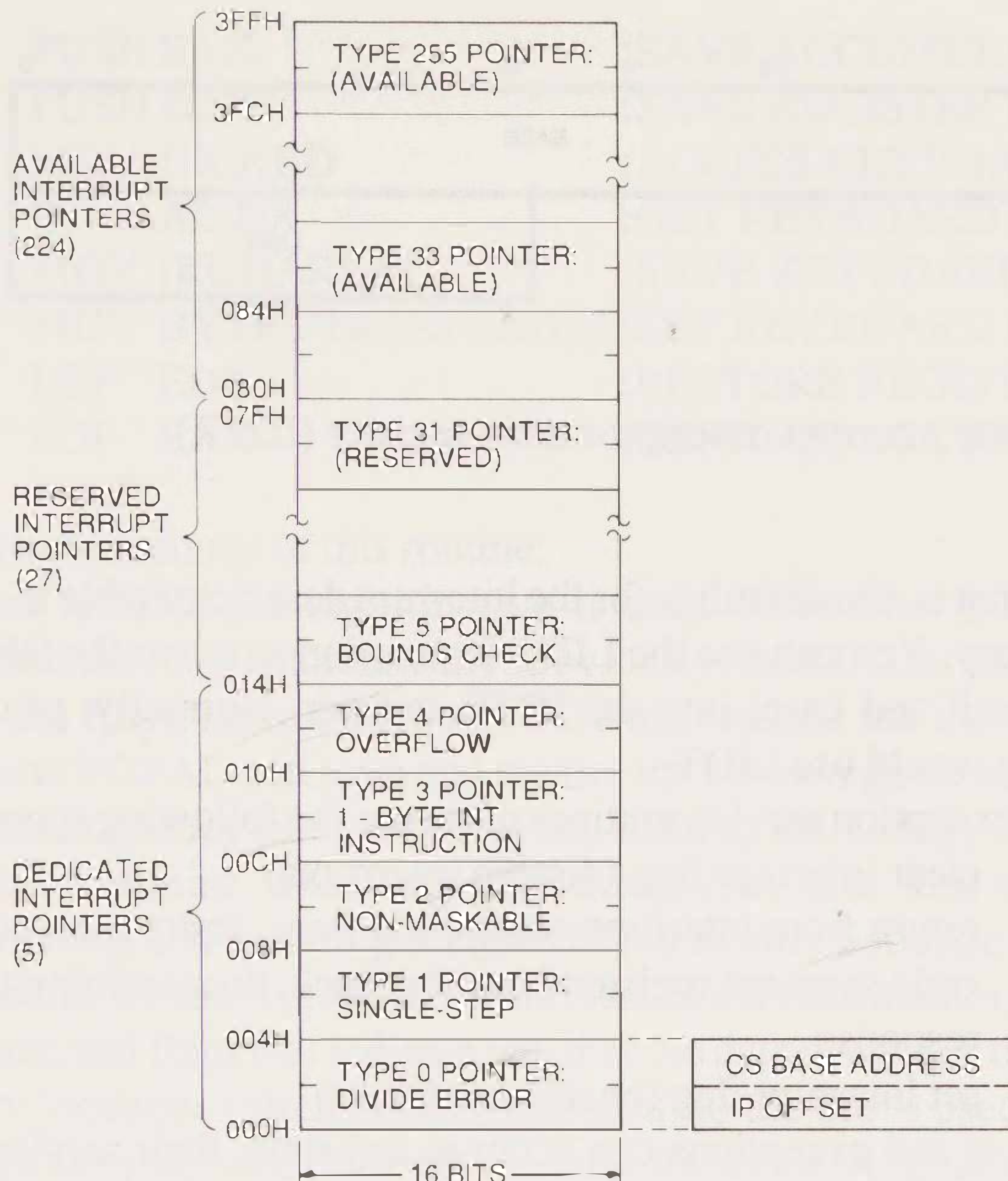


Figure 4-11

Memory map for 80386 interrupt vectors in an 8086 compatible mode.

- It gets new values for the code segment register and instruction pointer from memory. The locations used depend on the interrupt type and on the memory management approach (see Chapter 5). In the 8086-like modes, the locations are in a table called the *interrupt descriptor table*. Its base address is in the interrupt descriptor table (IDT) register. The offsets for interrupt type N are $4 \times N$ and $4 \times N + 1$ for the instruction pointer and $4 \times N + 2$ and $4 \times N + 3$ for the code segment register. Figure 4-11 shows the arrangement, starting at the base address.

The new factor here is the IDTR. It contains both a base address and a limit as shown in Figure 4-12. RESET makes the base address 0 and the limit 3FFH for compatibility

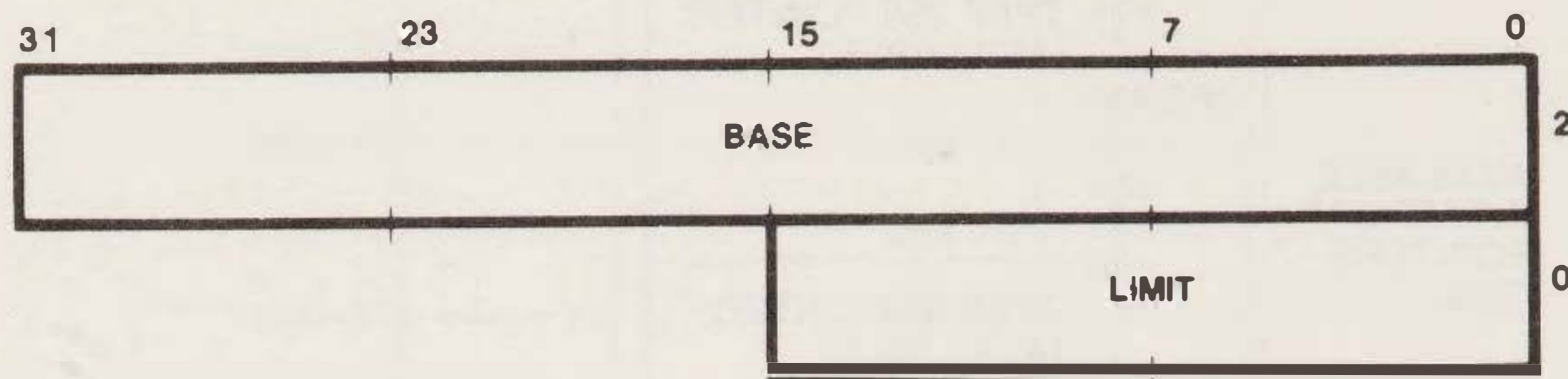


Figure 4-12

Data format for the interrupt descriptor table register (IDTR).

with the 8086. That is, the default is for the interrupt descriptor table to occupy the bottom 1K of memory. You can use the LIDT instruction to move the table. It loads a 6-byte operand (limit and base) into the IDTR register. Normally, of course, only an operating system would use LIDT.

Interrupt and exception service routines often use the following special instructions:

CLI — clear interrupt flag (disable interrupts)

IRET — return from interrupt; restore the flags, instruction pointer, and code segment register from the stack, thus undoing the interrupt response

STI — set interrupt flag (enable interrupts)

Since interrupts and exceptions can occur at any time, their service routines must run transparently. That is, they must not directly change any registers or flags that the main program may be using. Yet they must still do their job and indicate that it has been done properly. Their situation is like that of a janitor who must clean an office without moving anything or disturbing any work in progress. Typical instructions are, “Don’t touch any of those stacks of printouts. And don’t wake up the programmer who is sleeping in the middle of them. But be sure to vacuum properly (if you can see any carpet) and empty the wastebaskets (if you can find them).”

Service routines generally satisfy these requirements by saving registers in the stack initially. They must also use the stack or memory locations for temporary storage and results.

SAMPLE INTERRUPT SERVICE ROUTINES

1. Read a character from an interrupt-driven keyboard at port KBD. Put its value in memory location KCHAR and set memory location KFLAG to 1:

PUSHEAX	;SAVE ACCUMULATOR
PUSHEDX	;SAVE REGISTER EDX
MOV DX,KBD	;ACCESS KEYBOARD PORT
IN AL,DX	;GET KEYBOARD INPUT
MOV [KCHAR],AL	;SAVE KEYBOARD INPUT
MOV BYTE PTR [KFLAG],1	;SET KEYBOARD FLAG
POP EDX	;RESTORE REGISTER EDX
POP EAX	;RESTORE ACCUMULATOR
IRET	

Note the following features of this routine:

- a. It saves and restores the registers that it uses. Remember that the interrupt response saves the flags automatically. Obviously, most I/O service routines must save EAX and EDX, since the IN and OUT instructions use them. PUSHAD and POPAD can save and restore the entire register set. They may be useful even though they take extra time and stack space. Using them makes it unnecessary to determine which registers the routine uses. Nor do you have to change the saving and restoring sections if you modify a routine.
 - b. Information that the main program needs must be saved in memory. This includes pointers, data, and flags that indicate whether the data is valid. You may compare the memory locations (often called *mailboxes*) to the drops used in spy novels. Obviously, other programs must not use these locations. After all, you don't want your valuable information to end up in someone's junk mail or to be covered by the garbage from the local fish market.
 - c. The routine must restore registers in the opposite of the order in which it saved them. POPAD handles ordering automatically.
 - d. IRET ends the service routine and transfers control back where it was. It restores the flags as well as the instruction pointer and code segment register. Note that IRET reenables interrupts if they were enabled originally, since it restores the original flag values.
 - e. This unbuffered routine is like a peripheral with no local memory. If a second interrupt occurs before the first one is serviced, the data is simply lost. We could provide an overrun indicator by testing KFLAG before setting it.
 - f. The main program checks for a character by testing KFLAG. The only difference between this and polling is the activation by an interrupt.
2. Write a character to an interrupt-driven printer at port PRINTER. Get the character from the buffer addressed via a pointer at address PRBUFR. Increase the pointer

by 1 and decrease the count in location PRCNT by 1. If PRCNT is originally 0, put 1 in location PRFLG to indicate that the printer is ready but has not been serviced:

```

        PUSH EAX                ;SAVE ACCUMULATOR
        PUSH EDX                ;SAVE REGISTER EDX
        PUSH ESI                ;SAVE REGISTER ESI
        MOV AL,[PRCNT]          ;GET BUFFER LENGTH
        TEST AL,AL              ;TEST BUFFER LENGTH
        JZ   MARKPR             ;EXIT IF NOTHING IN BUFFER
        CLD                     ;SELECT
                                ; AUTOINCREMENTING

        MOV ESI,[PRBUFR]        ;GET BUFFER POINTER
        MOV DX,PRINTER          ;ACCESS PRINTER PORT
        OUTSB                   ;SEND DATA,
                                ; UPDATE POINTER

        MOV [PRBUFR],ESI        ;SAVE NEW BUFFER POINTER
        DEC BYTE PTR [PRCNT]    ;SUBTRACT 1 FROM BUFFER
                                ; LENGTH
        MOV BYTE PTR [PRFLG],0 ;INDICATE PRINTER SERVICED
                                ; IN CASE READY EARLIER

        JMP  ENDSR
MARKPR: MOV BYTE PTR [PRFLG],1 ;INDICATE PRINTER READY
ENDSR:  POP  ESI                ;RESTORE REGISTERS
        POP  EDX
        POP  EAX
        IRET

```

Note the following features of this routine:

- Output service routines must handle the case in which there is no data to send. This problem does not occur with input devices, since they always have data when they request service. One solution is to set a flag indicating that an interrupt has occurred but has not been serviced. The main program may then later send the data without waiting for an interrupt. In practice, of course, the service routine must also either clear or disable the interrupt.
- A buffered service routine acts much like a peripheral with its own local memory. The processor can put a large amount of data in the buffer, and the peripheral can then accept it at its own pace.
- A common approach in practice is for the buffer to have two pointers. One, the tail pointer, contains the address of the next empty location. The main program

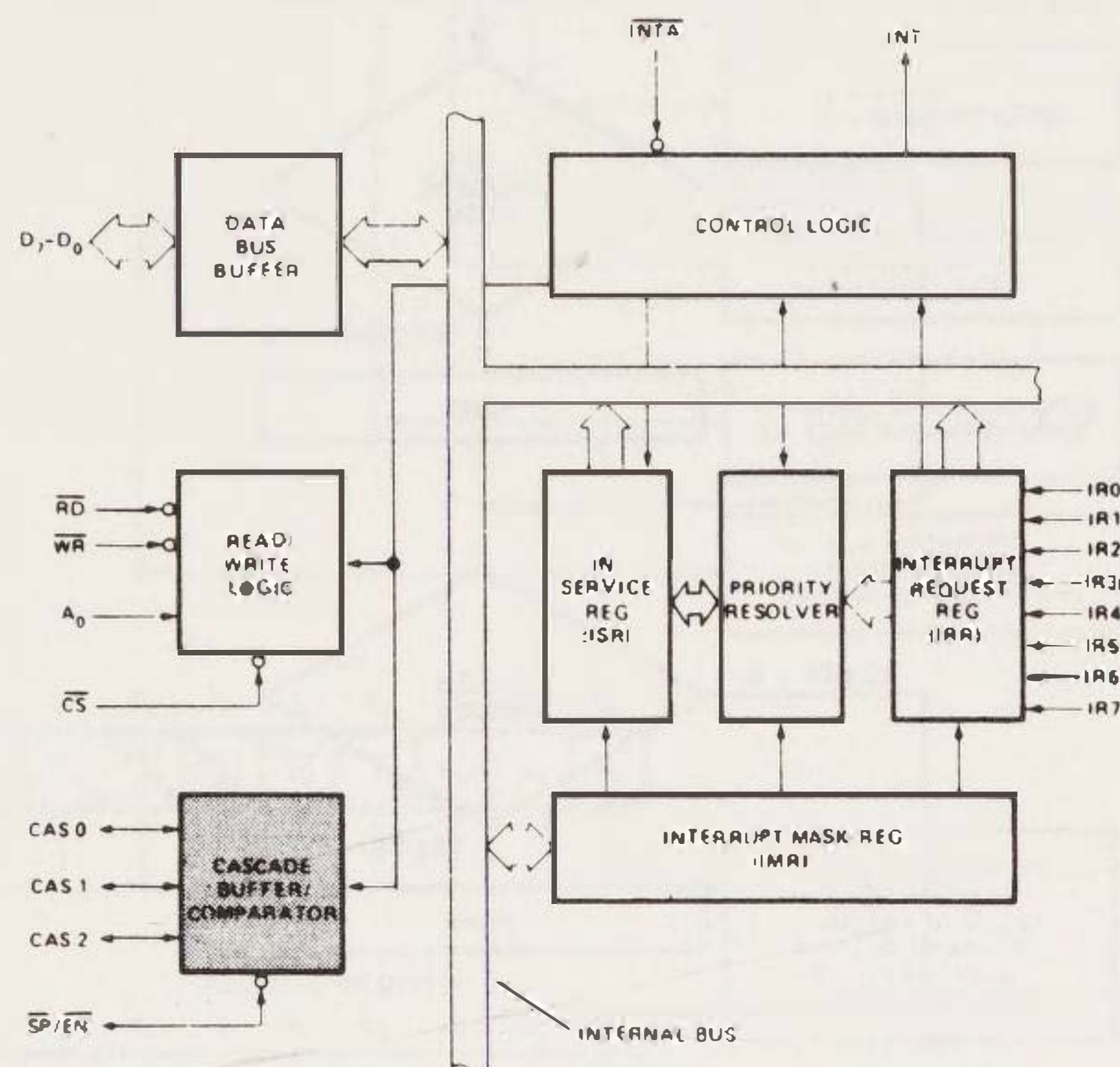


Figure 4-13

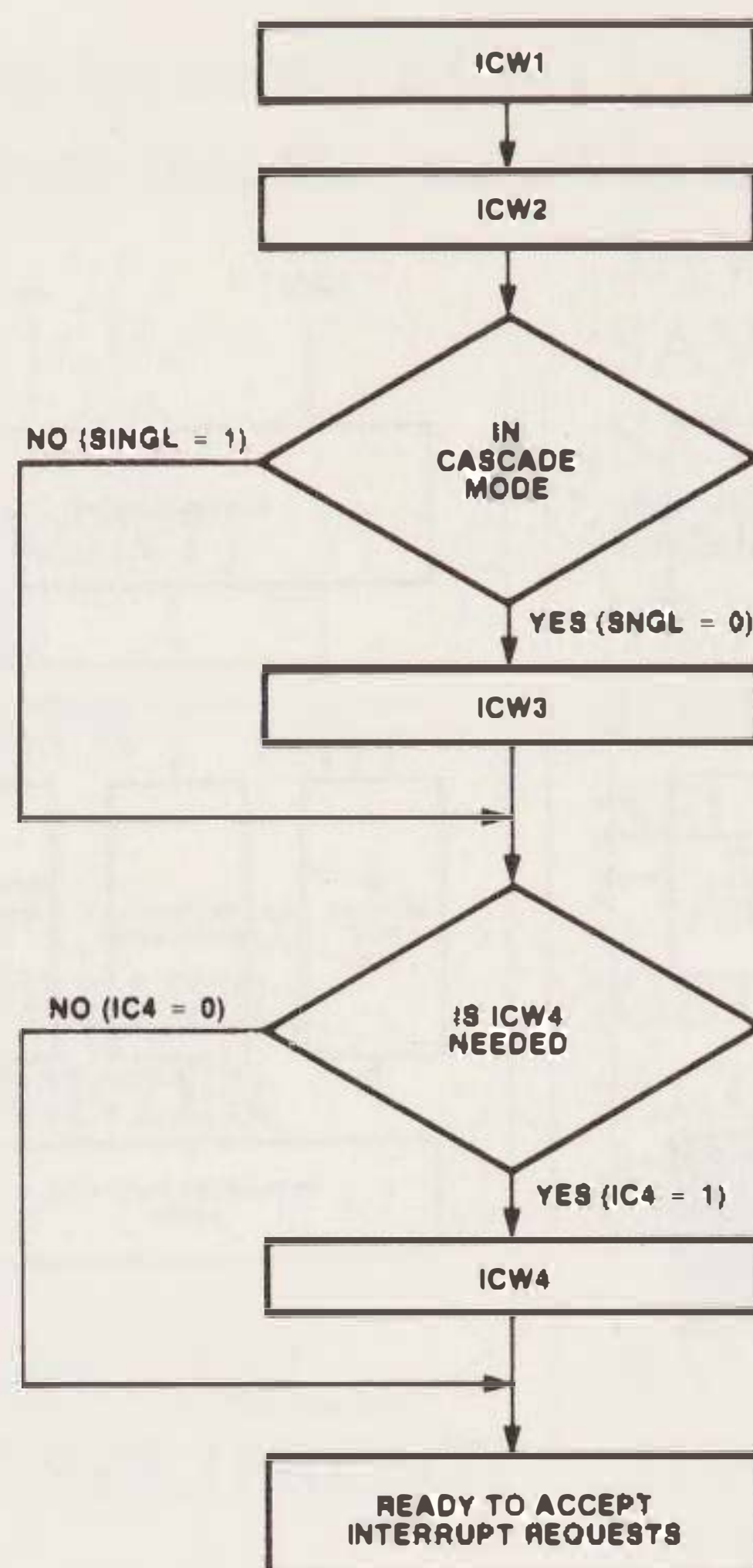
Block diagram of the 8259 Programmable Interrupt Controller (PIC). Reprinted courtesy of Intel Corporation, Santa Clara, California.

uses this pointer to put characters in the buffer. The other, the head pointer, contains the address of the oldest filled location. The service routine uses this pointer to send characters from the buffer. The part of the buffer filled extends from the head pointer to just before the tail pointer. This part can be anywhere physically. It can even extend from the end of the buffer back past the beginning (like a television picture with the vertical hold off kilter). This feature (called *wraparound*) applies if the routines set the pointers back to the base address when they reach the end of the buffer.

INTERRUPT CONTROLLER

In practice, interrupt systems require external hardware. This hardware must:

- Generate a single interrupt signal from many inputs

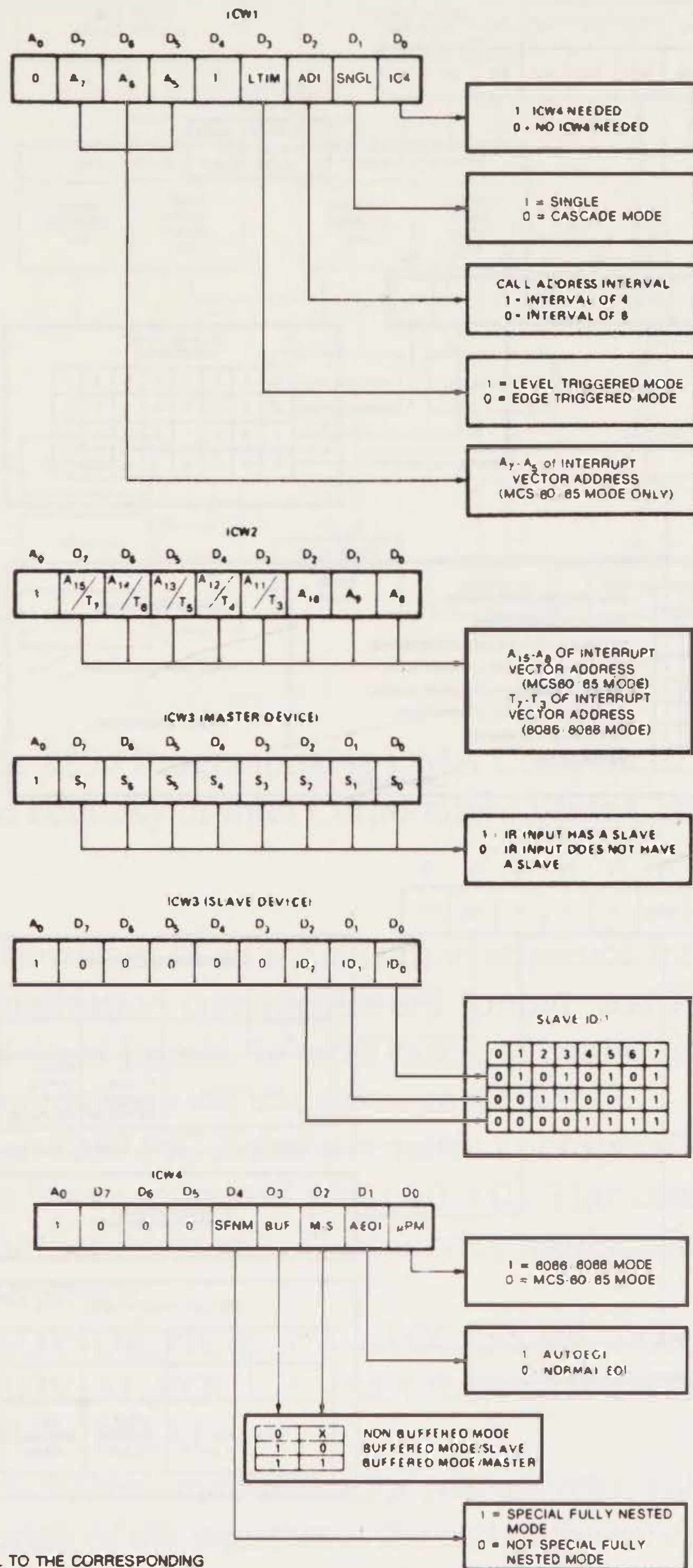
**Figure 4-14**

Initialization sequence for the 8259 Programmable Interrupt Controller (PIC). Reprinted courtesy of Intel Corporation, Santa Clara, California.

- Select the input that has the highest priority for recognition
- Provide the interrupt type (often called *vector*) when the CPU asks for it

The device generally used for this purpose in 80386-based computers is the 8259 programmable interrupt controller or PIC. Figure 4-13 is a block diagram of the 8259 PIC. Its main features are the following:

- An 8-level priority controller (expandable to 64 levels by adding more devices).
- Individual request masks. They allow you to block any input at any time.
- Variety of programmable operating modes including fully nested, automatic rotation, and specific rotation. Rotating priorities help avoid the situation in which a low-priority interrupt is ignored virtually forever because of a series of high-priority interrupts. The predicament is like that of a telephone caller who is left on hold indefinitely, listening to some awful recorded music.



NOTE 1: SLAVE ID IS EQUAL TO THE CORRESPONDING MASTER IR INPUT.

Figure 4-15

Initialization command word (ICW) format for the 8259 Programmable Interrupt Controller (PIC). Reprinted courtesy of Intel Corporation, Santa Clara, California.

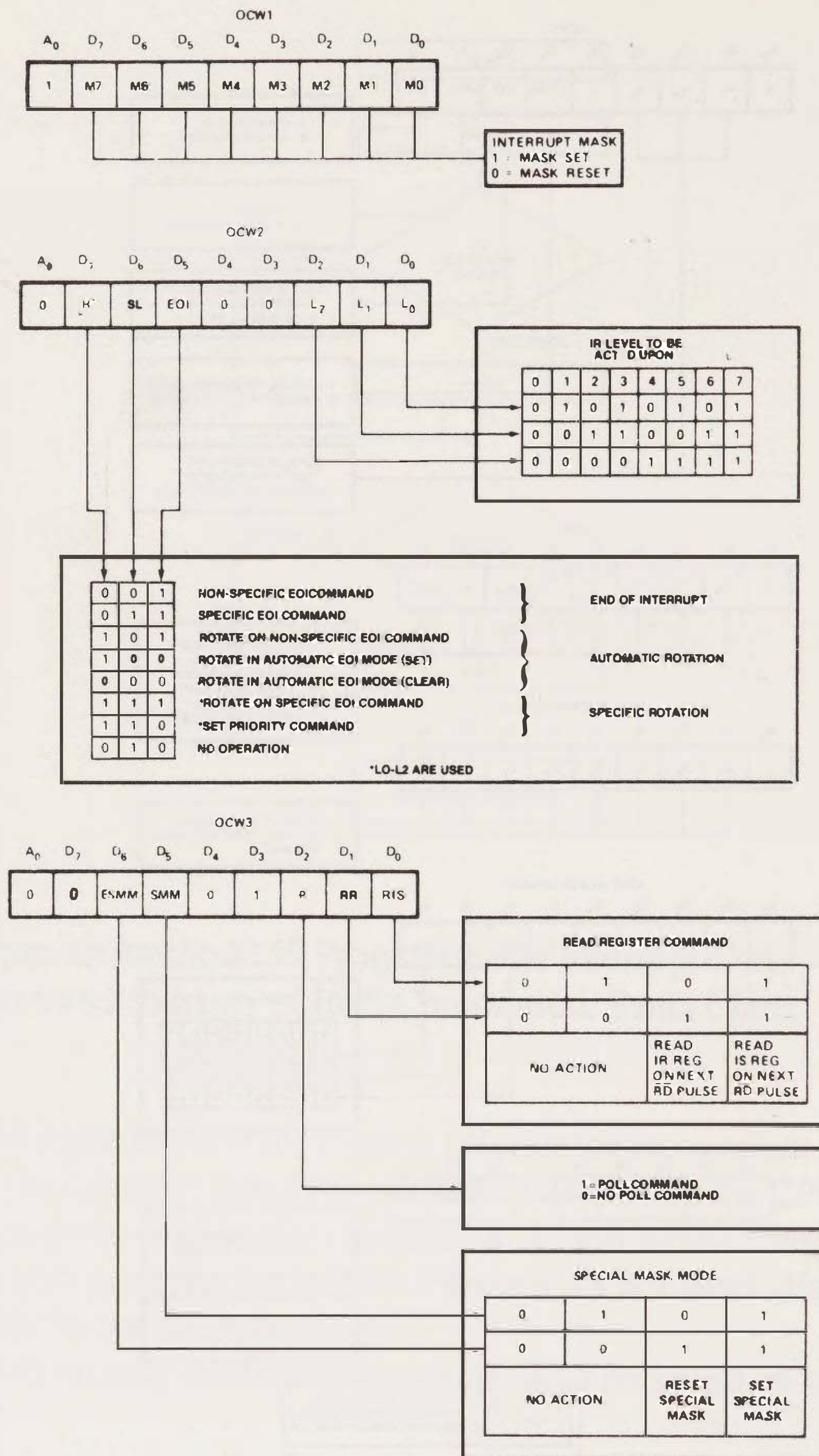


Figure 4-16

Operation command word (OCW) format for the 8259 Programmable Interrupt Controller (PIC). Reprinted courtesy of Intel Corporation, Santa Clara, California.

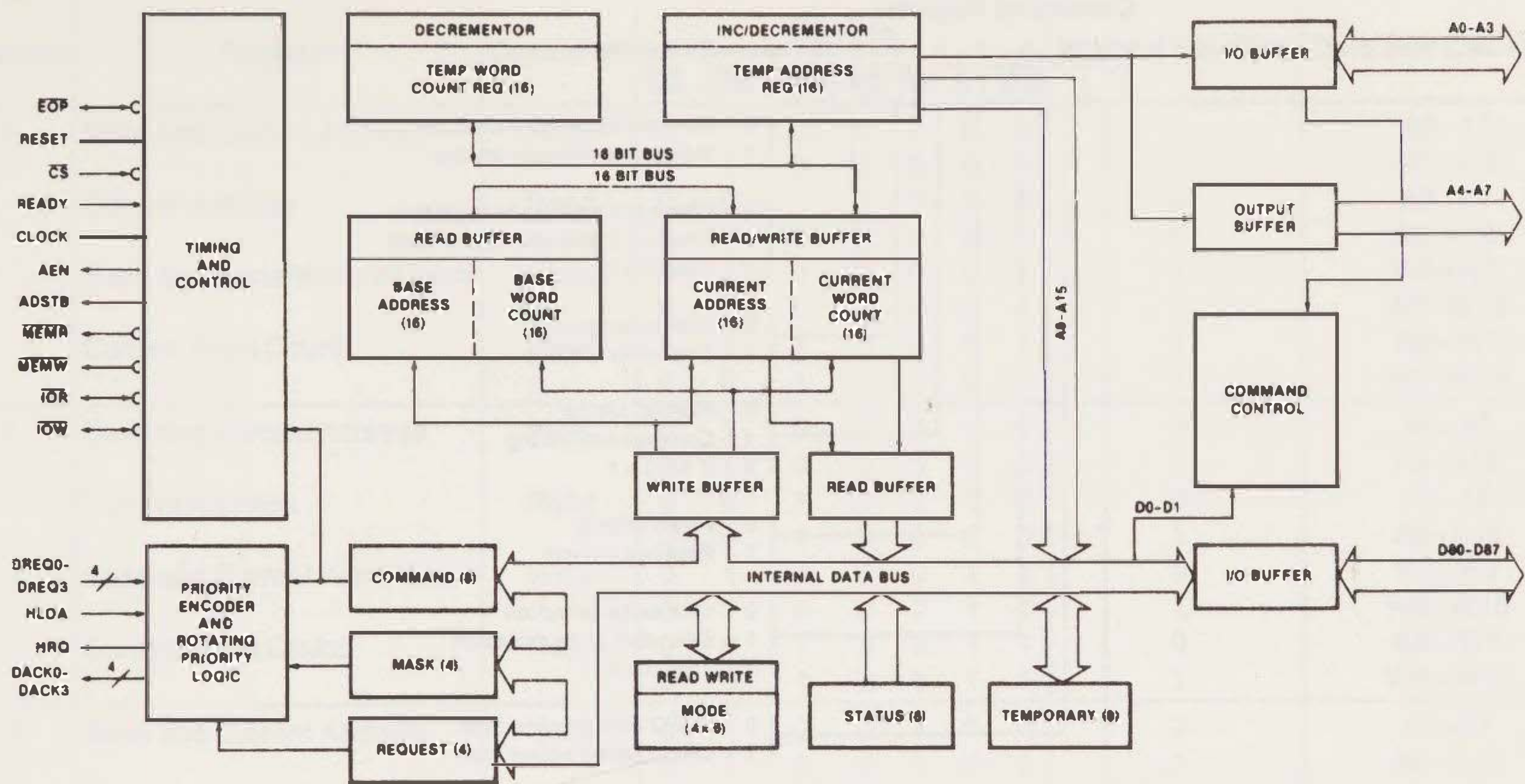


Figure 4-17

Block diagram of the 8237 Programmable DMA Controller (DMAC). Reprinted courtesy of Intel Corporation, Santa Clara, California.

Figure 4-14 is a flowchart of an initialization sequence for an 8259 PIC. Figure 4-15 contains the initialization command word format, and Figure 4-16 contains the operation command word format. In most cases, of course, only an operating system or overall I/O startup routine ever initializes an 8259 PIC.

A key feature of an 8259 PIC is that it requires a nonspecific end-of-interrupt (EOI) command sent to its lower addressed port (A0 = 0). This command clears the In Service bit, allowing further interrupts. Thus service routines for 8259-based interrupts must end with the sequence

```
MOV DX,PIC0      ;ACCESS PIC PORT
MOV AL,EOI       ;CLEAR 8259 INTERRUPT
OUT DX,AL
```

The EOI command is 20 hex, and PIC0 is the port address. This sequence should come just before the restoring of the registers at the end of the service routine. Sending EOI commands is the only contact most applications programmers have with an 8259 PIC. Beware, however — service routines for most computers will not work without an EOI command at the end.

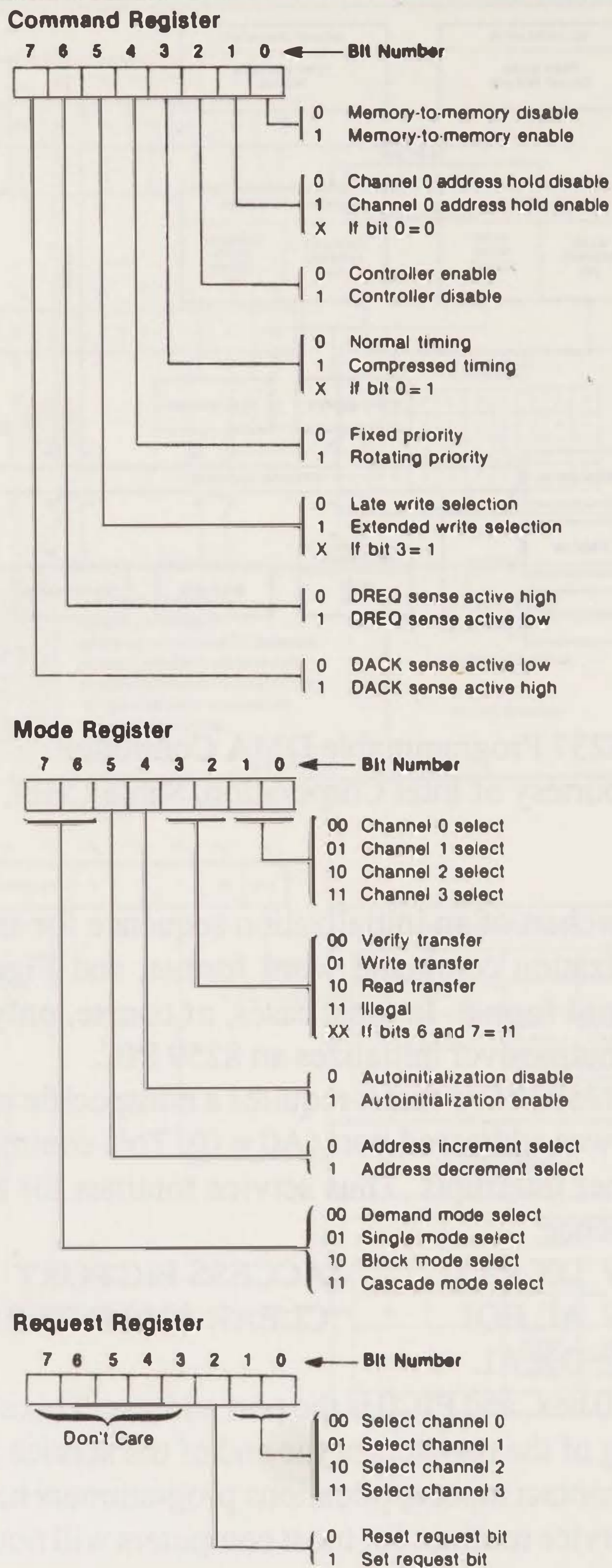


Figure 4-18

8237 DMAC Command, Mode, and Request registers. Reprinted courtesy of Intel Corporation, Santa Clara, California.

Input/Output

Channel	Register	Operation	Signals							Internal Flip-Flop	Data Bus DB0-DB7
			CS	IOR	IOW	A3	A2	A1	A0		
0	Base and Current Address	Write	0	1	0	0	0	0	0	0	A0-A7
			0	1	0	0	0	0	0	1	A8-A15
	Current Address	Read	0	0	1	0	0	0	0	0	A0-A7
			0	0	1	0	0	0	0	1	A8-A15
	Base and Current Word Count	Write	0	1	0	0	0	0	1	0	W0-W7
			0	1	0	0	0	0	1	1	W8-W15
	Current Word Count	Read	0	0	1	0	0	0	1	0	W0-W7
			0	0	1	0	0	0	1	1	W8-W15
1	Base and Current Address	Write	0	1	0	0	0	1	0	0	A0-A7
			0	1	0	0	0	1	0	1	A8-A15
	Current Address	Read	0	0	1	0	0	1	0	0	A0-A7
			0	0	1	0	0	1	0	1	A8-A15
	Base and Current Word Count	Write	0	1	0	0	0	1	1	0	W0-W7
			0	1	0	0	0	1	1	1	W8-W15
	Current Word Count	Read	0	0	1	0	0	1	1	0	W0-W7
			0	0	1	0	0	1	1	1	W8-W15
2	Base and Current Address	Write	0	1	0	0	1	0	0	0	A0-A7
			0	1	0	0	1	0	0	1	A8-A15
	Current Address	Read	0	0	1	0	1	0	0	0	A0-A7
			0	0	1	0	1	0	0	1	A8-A15
	Base and Current Word Count	Write	0	1	0	0	1	0	1	0	W0-W7
			0	1	0	0	1	0	1	1	W8-W15
	Current Word Count	Read	0	0	1	0	1	0	1	0	W0-W7
			0	0	1	0	1	0	1	1	W8-W15
3	Base and Current Address	Write	0	1	0	0	1	1	0	0	A0-A7
			0	1	0	0	1	1	0	1	A8-A15
	Current Address	Read	0	0	1	0	1	1	0	0	A0-A7
			0	0	1	0	1	1	0	1	A8-A15
	Base and Current Word Count	Write	0	1	0	0	1	1	1	0	W0-W7
			0	1	0	0	1	1	1	1	W8-W15
	Current Word Count	Read	0	0	1	0	1	1	1	0	W0-W7
			0	0	1	0	1	1	1	1	W8-W15

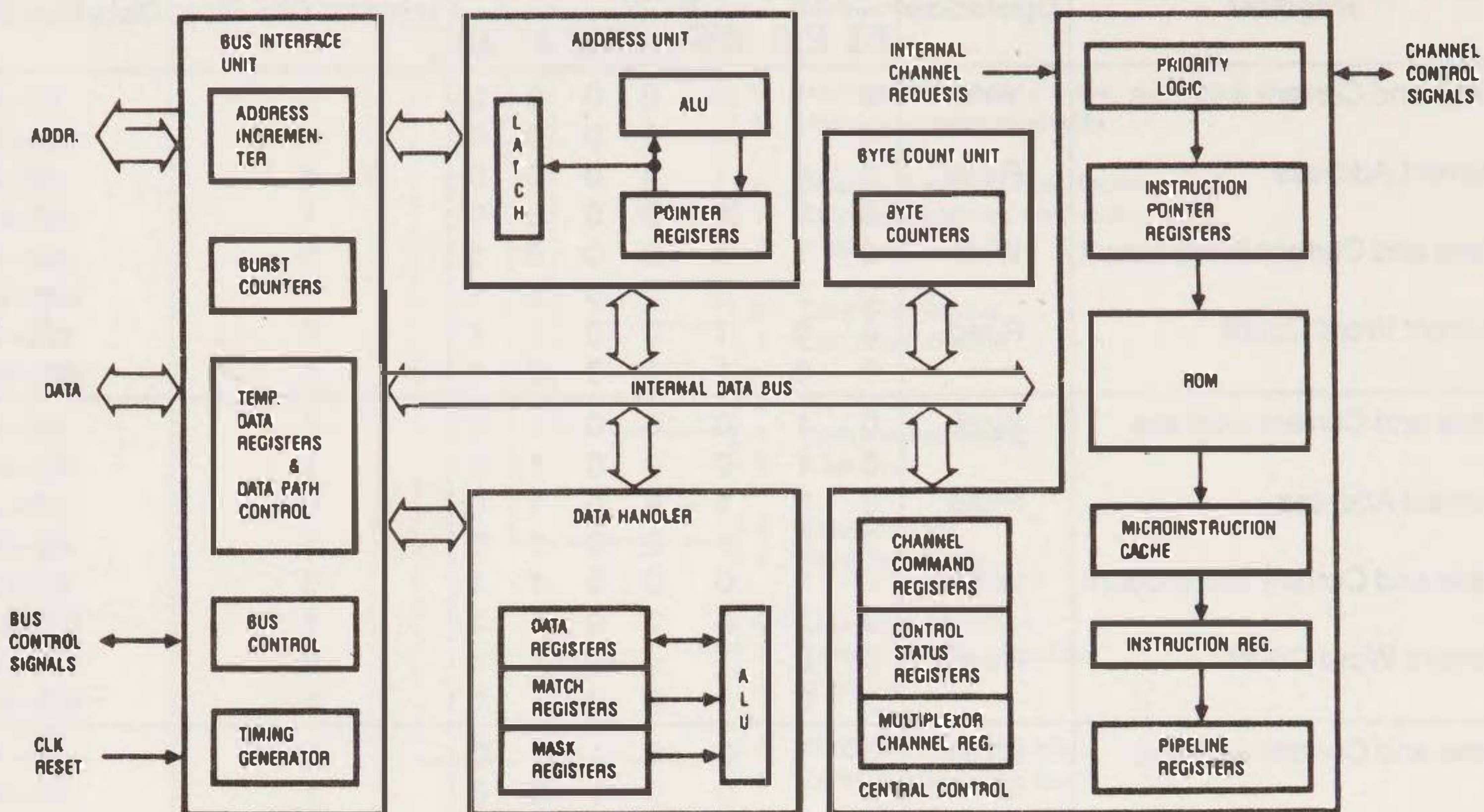
Figure 4-19

Word count and address register command codes for the 8237 programmable DMA controller. Reprinted courtesy of Intel Corporation, Santa Clara, California.

DIRECT MEMORY ACCESS

Direct memory access is the fastest I/O method, but it requires the most external hardware. An external controller must:

- Manage memory addressing.
- Keep track of the number of transfers performed.
- Control the CPU while the transfers occur. This involves forcing the CPU to enter a suspended state and stay in it until the operation ends.

**Figure 4-20**

Block diagram of the 82258 advanced direct memory access coprocessor (ADMA).

Many older computers use the 8237 programmable DMA controller (DMAC). This device (see Figure 4-17 for a block diagram) has the following features:

- Four independent DMA channels
- Independent control of DMA requests and channel initialization
- Single transfer, block transfer, or demand transfer operating modes

Figure 4-18 shows the 8237 DMAC's command, mode, and request registers. Figure 4-19 lists its command codes.

Newer systems with larger address spaces use the 82258 advanced direct memory access coprocessor (ADMA). This device also has four independently programmable channels. However, it also provides for chaining of commands and data, thus permitting independent processing and allowing data blocks to be scattered anywhere in memory. Figure 4-20 is a block diagram of the 82258 ADMA. This device is virtually a CPU on its own rather than just a simple controller.

Most 80386 I/O sections use programmable I/O chips to provide flexibility and to save on power and board space. Common chips include the 8250 serial interface (ACE), the 8255 parallel interface (PPI), and the 8253 or 8254 timer (PIT). These devices can handle a wide range of applications, but there are no standards for their use or programming.

Interrupt-driven I/O depends on the 80386's nonmaskable and maskable interrupt inputs. Maskable interrupts require additional identifying information (called a vector or interrupt type). In response to these inputs, the 80386 saves the flags, instruction pointer, and code segment register in the stack. It then obtains new instruction pointer and code segment register values from an interrupt descriptor table in memory. The interrupt descriptor table register contains the table's base address and upper limit.

80386 Memory Management

*One must have a good memory to be able
to keep the promises one makes.*

Nietzsche, Human, All Too Human

*A very fair scholar I was too; no thought
but a great memory.*

Beckett

This chapter describes 80386 memory management techniques. It first explains the processor's operating modes and then covers segmentation, paging, memory protection, and initialization of memory management systems.

80386 HIGHLIGHTS

The 80386's new operating mode is the virtual 8086 mode. It lets 8086 programs run simultaneously with programs that use advanced 80386 features such as protection, paging, and very large segments. It bridges the gap between popular 8086-based (particularly MS-DOS) software and the new features of the 80386's protected mode.

The 80386's major new memory management features are:

- It allows segments as large as 4 Gb (about 4 billion bytes). This compares with the 64-Kb limit on previous processors. The result is that almost any applications program can fit in a single 80386 segment. There is no need to separate program and data areas or to divide areas into segments. Segmentation can thus be practically invisible on an 80386. It does, however, provide a convenient platform for implementing protection.
- It provides on-chip support for paging. The 80386 can divide logical memory into 4-Kb pages. It will then automatically use a page directory and a page table to look up the physical mapping of an address and determine whether it is currently in memory or on disk. The 80386 also has an on-chip cache that holds the most recently accessed pages for the current task. Demand-paging thus becomes both fast and easy to implement.

Of course, access to these features requires a 386-specific operating system such as XENIX V/386 (Microsoft) or System V/386 (Intel). Access to the virtual 8086 mode requires a virtual 8086 monitor such as VP/ix (Interactive Systems) or OS/Merge 386 (Locus Computing). 386-specific features are not accessible under an operating system that must also run on 8086/8088-based computers (MS-DOS) or on 80286-based computers (OS/2).

MEMORY MANAGEMENT

The idea of memory management is relatively new to personal computers. Large computers, on the other hand, have used it for decades. Its aim is to make efficient use of a memory section consisting of:

- High-speed, expensive memory (called a *cache*)
- Standard (main) memory
- Backup (secondary) storage, usually magnetic disk

Figure 5-1 shows a typical memory section with a memory management unit (MMU). The MMU converts addresses as the program sees them (logical addresses) into those needed to access actual memory locations (physical addresses). The conversion is almost invisible as far as programmers are concerned. You may compare the MMU to the electromechanical system that converts an automobile driver's control actions into the actual mechanical operations that run the car. Most drivers neither understand the conversion nor care about how it works.

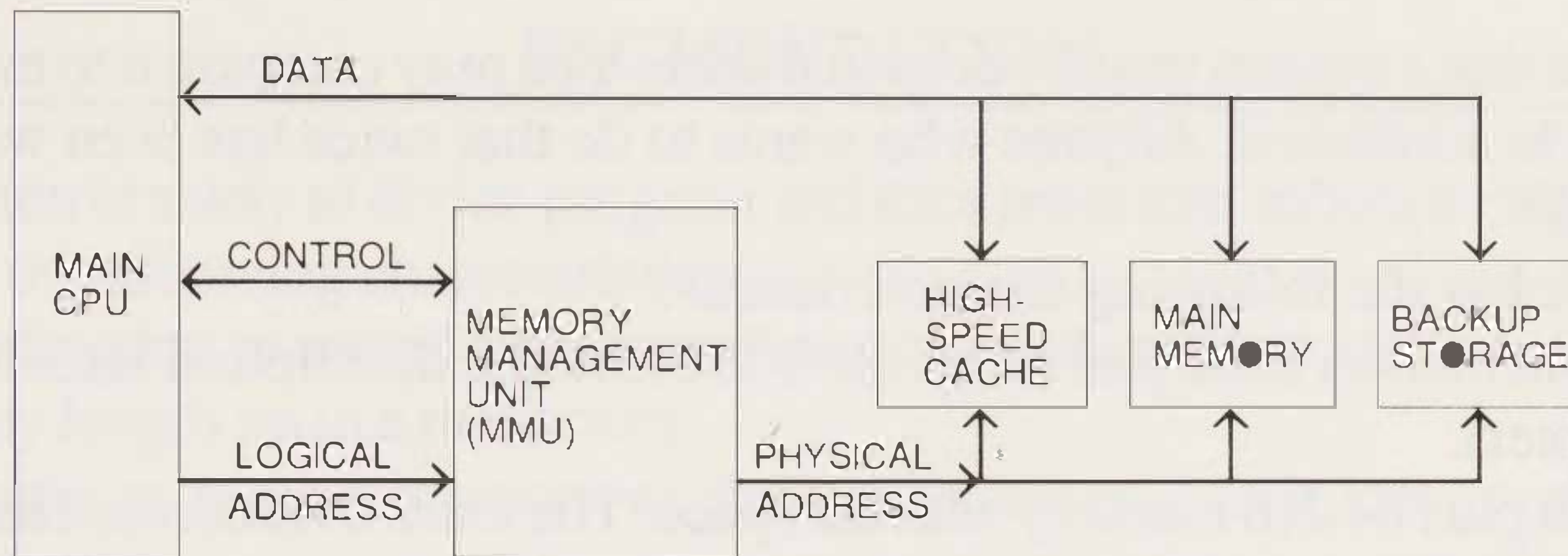


Figure 5-1

A memory section with a memory management unit.

The MMU also allows any reasonable division of the memory section into cache, main memory, and secondary storage. The tradeoff here is obvious. Faster storage costs more per unit. A computer with more fast memory will execute programs faster than one that must use main memory and secondary storage more often. The management process itself takes some time besides, although the 80386 masks this by doing it in parallel with other operations.

The 80386 has an on-chip MMU, as do the 80286 and Motorola 68030. On the other hand, the Motorola 68000, 68010, and 68020 use a separate MMU. The tradeoff here is also simple. An on-chip MMU is faster but cannot have all the features of a separate device.

The rest of this chapter describes 80386 memory management. Chapter 8 discusses cache memory.

OPERATING MODES

The 80386 can operate in either of two main modes:

1. Real mode
2. Protected (virtual) mode

In the real mode, used mostly for startup, the 80386 acts like a 32-bit version of the 8086 processor. In the protected mode, the usual operating mode, the 80386 acts like a 32-bit version of the 80286 processor. The choice between the two modes depends on bit 0 of control register 0 (the protection enable or PE bit). The selection is system-wide, rather than on a task-by-task basis. Changing from real to protected mode is a

drastic action that a system usually does just once. You may compare it to moving from adolescence to adulthood. Anyone who wants to do that twice has been watching too many movies!

Real mode has the following characteristics:

- Segmentation done just as on the 8086 (see the description later in this chapter).
- 1 Mb plus 64-Kb memory address space. The extra 64K comes from the carry in the segmentation calculation. The 8086 simply drops the carry, so its address space is exactly 1 Mb.
- Effective addresses (within a segment) limited to 64K.
- Operands and addresses are 16 bits by default. However, you can apply 32-bit overrides to either or both.

Real mode is thus not exactly like an 8086, as the 80386's 32-bit capabilities and new applications-oriented instructions remain available. You would need to use either assembly language or a special compiler with an 80386 mode to access them.

In protected mode, the 80386 has a submode called *virtual 8086 mode*. It is selected by setting the Virtual Mode (VM) flag, bit 17 of the extended flags (Fig. 2-3), to 1. In V86 mode, as in real mode, the 80386 acts much like an 8086. The difference between the modes is that the processor can enter V86 mode on a task basis. That is, any task that initializes the extended flags can also choose between V86 mode and protected mode. Thus a system can run, either "simultaneously" (under multitasking) or sequentially, both 8086 programs in V86 mode and generalized 80386 programs in protected mode. The switch to V86 mode has no drastic effects on memory management or address generation. You may compare it to playing juvenile games without going through a time warp.

Note that there is no virtual 80286 mode, perhaps reflecting the small amount of unique 80286 software. In practice, the 80386 can run 80286 code directly with a few minor exceptions. However, the programmer must produce 80286 emulation by limiting address and operand size, trapping or restricting the use of 80386 features, and avoiding new 80386 instructions. For example, this is necessary to create programs that will run under OS/2 or XENIX V/286.

OS/2 can switch back and forth between real and protected modes. This is how it runs MS-DOS applications. It must use real mode rather than virtual 8086 mode so that it can run on 80286-based computers. The problem here is that real mode provides no protection from ill-behaved applications. They can manipulate hardware directly, change interrupt vectors, or alter operating system tables or parameters. The result is like a secure installation that has occasional "open houses."

SEGMENTATION

Segmentation is a way to divide program and data areas into subunits called *segments*. They have the following characteristics:

- Dedication to specific program functions such as code, data, or stack
- Any length up to a maximum
- Apply to logical memory as the programmer sees it, not to physical memory

The 80386 lets a program access up to six segments at a time through the following registers:

Code segment (CS), the area from which the processor is currently fetching and executing instructions

Stack segment (SS), the area containing the hardware stack used by subroutines and interrupts

Data segments (DS, ES, FS, and GS), areas used for temporary data storage. The only new 80386 features here are two more data segment registers (FS and GS). The accessible segments are like open files in a database management system.

8086 Segmentation Methods

8086 segments may be up to 64 Kb long. A 16-bit number defines a segment. Its base address is that number times 16. For example, segment 9000 hex starts at address 90000 hex. Segments thus always start at addresses divisible by 16.

The 8086 translates logical addresses into physical addresses as follows:

1. Multiply the segment number by 16.
2. Add the address within the segment (called the *offset*) to the product.

The translation is the same on all 8086/8088-based computers. It is independent of the operating system or any other software.

Let us see how this simple one-stage mapping function works in actual examples:

1. Suppose an instruction gets data from address (offset) E137 hex and the data segment register contains 1800 hex. The data's physical address is

$$18000 + E137 = 26137$$

Note that multiplying a hex number by 16 is like multiplying a decimal number by 10. After all, 16 is 10 hex. This is obvious if you come from a computerized planet

Table 5-1
Default segment register selection rules

Memory Reference Needed	Segment Register Used	Implicit Segment Selection Rule
Instructions	Code (CS)	Automatic with instruction prefetch
Stack	Stack (SS)	All stack pushes and pops. Any memory reference that uses ESP or EBP as a base register.
Local Data	Data (DS)	All data references except when relative to stack or string destination.
Destination Strings	Extra (ES)	Destination of string instructions.

where people have 16 fingers. A easy way to multiply by 16 is with a 4-bit (1 hex digit) logical left shift.

- Suppose an instruction starts at address (offset) 4A5B hex and the code segment register contains C000 hex. The instruction's physical address is
 $C0000 + 4A5B = C4A5B$

Each instruction has a default segment register assignment (see Table 5-1). You can override the default generally by prefixing an instruction and specifying a different segment register. For example, the instruction

```
MOV     EAX,[OPER]
```

loads data from offset OPER in the segment defined by DS. Similar moves with segment overrides are

```
MOV     EAX,ES:[OPER]
```

```
MOV     EAX,CS:[OPER]
```

The first instruction gets data from the segment defined by ES, the second one from the code segment.

Overrides apply even to instructions with implied memory operands as long as you use a form that allows for them. For example, the instructions

XLAT

and

XLAT [EBX]

both do a table lookup in the data segment. The [EBX] serves no purpose, as XLAT uses EBX automatically and does not allow any other address. However, [EBX] is essential if you want to use a different segment. For example, the instruction

XLAT CS:[EBX]

uses a table in the code segment. Without the [EBX], there would be no place to put the segment override. Think of it as keeping a hitching post in front of your house, just in case someone arrives on horseback.

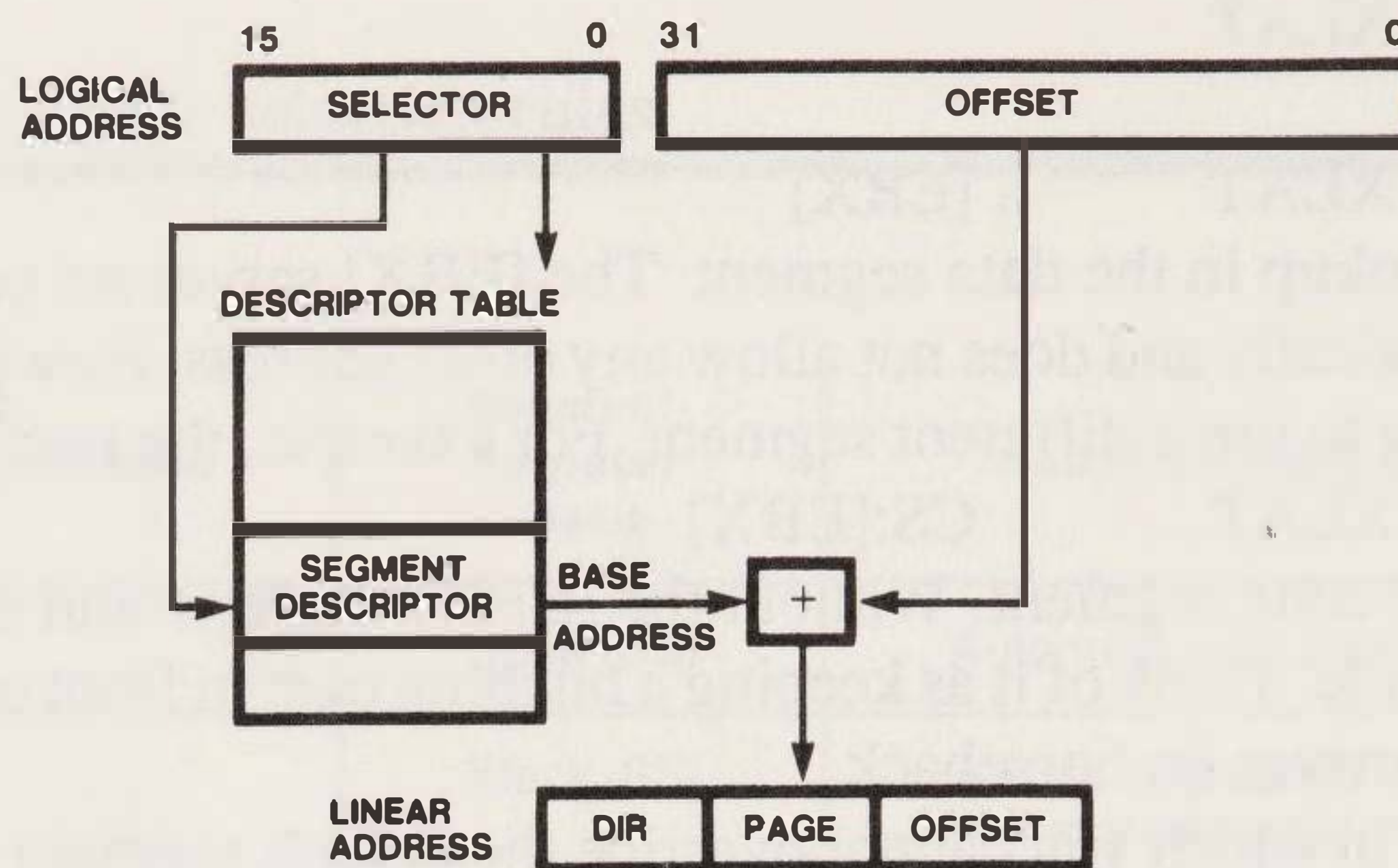
The only cases in which you cannot override the default segment assignment are:

- The use of ES for destination addresses (accessed via DI) in the string primitives CMPS, INS, MOVS, SCAS, and STOS. Note that you can override the source segment assignment, but not the destination. As Dr. Seuss says about such funny things, “Don’t ask me why. Go ask your mother.”
- The use of SS in stack instructions (PUSH, POP, CALL, RET, and so on).
- The use of CS for instruction fetches.

Segmentation on the 8086 is thus a simple way to extend the addressing range of 16-bit operands. The 8086 computes 20-bit addresses (allowing access to 1M of memory) from two 16-bit components that it can manipulate easily. There is no time penalty for segmentation, as it occurs in parallel with instruction execution and uses separate arithmetic facilities.

Segmentation does, however, create some awkward problems. How do you handle programs or data areas larger than 64 Kb? What happens when you reach the end of a segment? There are, in general, no easy answers to these questions. Segmentation will therefore never be programmers’ favorite approach to memory management. In practice, many compilers use a single-segment approach (called a *small memory model*) that limits code and data to 64 Kb each. When code or data may be larger, programs may run much more slowly under the multiple-segment *large memory model*. The slowdown is the result of the time spent checking for the ends of segments and changing the set of accessible segments.

The simple mapping procedure makes it easy to divide memory into consecutive nonoverlapping segments. For example, we could divide the entire 1-Mb address space into 16 nonoverlapping 64-Kb segments. The first one would be segment 0, the second segment 1000 (starting at address 10000), the third segment 2000 (starting at address

**Figure 5-2**

How a selector is used to compute a linear address.

20000), etc. This is a common way to describe the memory of an IBM PC or PC clone. A program that needs more than 64 Kb for code or data can readily compute new segment register values needed to access consecutive addresses. Many 8086-based compilers use this approach to handle large programs or data areas.

For example, suppose that a program has just accessed the data word at offset FFFE of segment 7000 hex. To reach the next higher address, assuming that the offset is in a base register, the program must:

1. Add 2 to the base register.
2. If the result is 0, add 1000 hex to the data segment register. This requires some MOVs, as arithmetic instructions cannot operate on segment registers.

The next higher address is address 0 of segment 8000 hex. Its physical address is 80000 hex. Note that checking for a zero base register value increases the execution time of each access.

DESCRIPTORS USED FOR SPECIAL SYSTEM SEGMENTS

The diagram illustrates the structure of a 32-bit segment descriptor. The top row shows the bit fields: BASE 31..24 (bits 31-24), G (bit 23), X (bit 22), O (bit 21), AVL (bit 20), LIMIT 19..16 (bits 19-16), P (bit 15), DPL (bits 14-13), 1 (bit 12), TYPE (bits 11-8), A (bit 7), and BASE 23..16 (bits 23-16). The bottom row shows two 16-bit fields: SEGMENT BASE 15..0 (bits 15-0) and SEGMENT LIMIT 15..0 (bits 15-0).

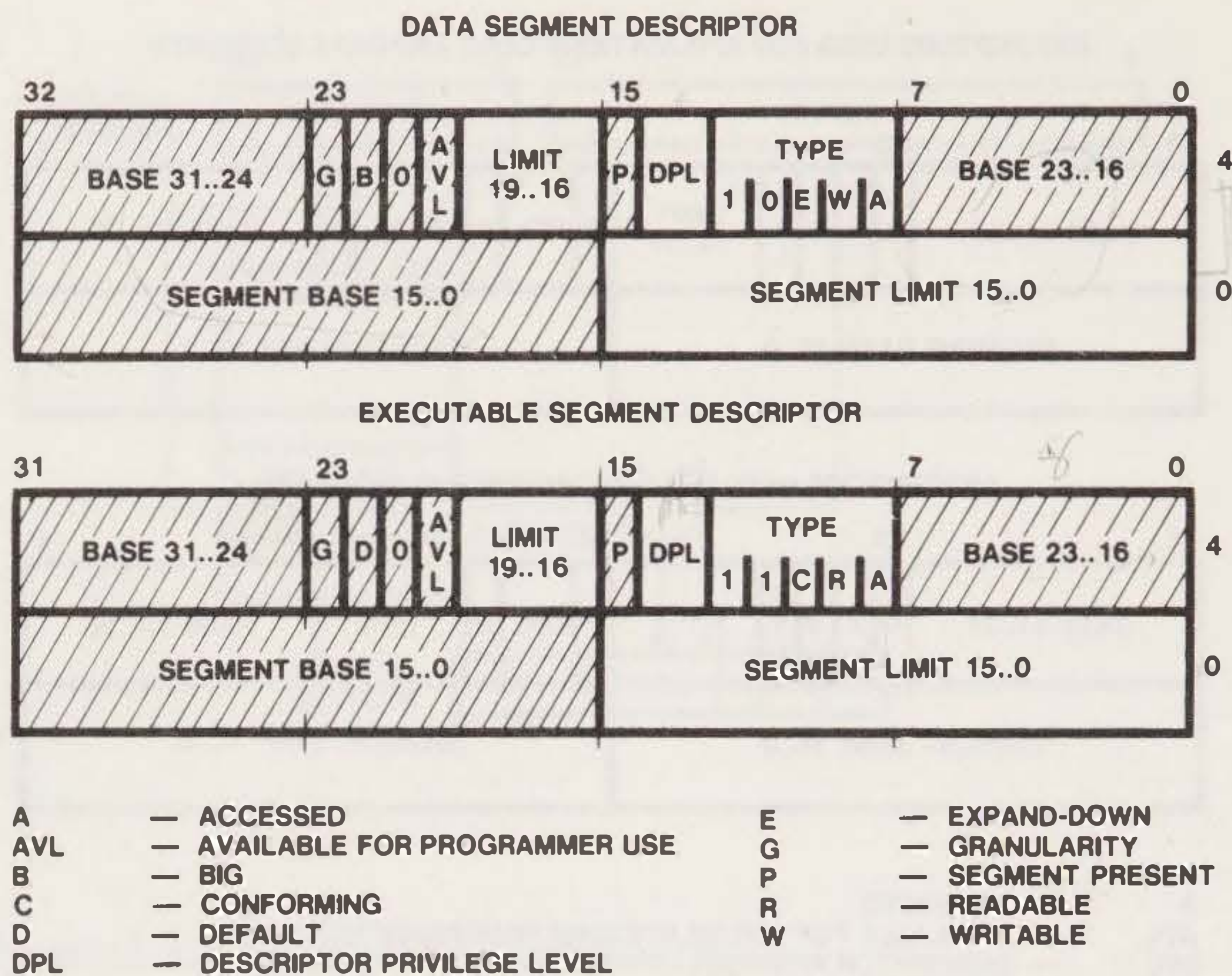
Figure 5-3
General format for segment descriptors.

Protected Mode Segmentation

In the 80386's protected mode, segmentation works differently than on an 8086. Here, a segment register contains a *selector*. The selector then points to a *descriptor* stored in a table in memory. The *descriptor*, in turn, contains the segment's base address, limit, and other attributes. Figure 5-2 shows how a selector points to a descriptor.

Figure 5-3 contains the general format of segment descriptors. The fields are:

- Base address (32 bits). The processor uses it to compute an intermediate result called a *linear address*, much as the 8086 uses the segment number times 16. That is, the processor adds the offset and base address. Note that no multiplication or shifting is necessary, and a segment can begin anywhere.
- Segment limit (20 bits). This defines the segment's length.
- Type (5 bits). Figure 5-4 shows the type field for data and code (executable) segments. We will discuss other types of descriptors later in this chapter and in Chapters 6 and 7.

**Figure 5-4**

Format for data segment and executable (code) segment descriptors.

- Descriptor privilege level (2 bits). It indicates how trusted a code segment is or how trusted it must be to use the descriptor. We will discuss privilege levels later in this chapter.
- A segment present (P) bit. An operating system can use this bit to implement virtual memory at the segment level. In practice, most systems implement virtual memory through paging instead. A major reason is that the variable size of segments makes swapping time consuming and difficult to implement.
- An accessed (A) bit. The processor sets A whenever it accesses the segment. An operating system could use it together with the P bit to implement virtual memory. A common approach is to test and clear the A bits regularly. The OS can then use the counts of how often each A bit was set to identify segments that have not been used recently. Those segments are obvious candidates for removal from physical memory.

- A granularity (G) bit. It specifies whether the limit is in units of bytes (0) or 4 Kb (1). It thus allows a 20-bit limit field to specify segments as large as 4 Gb. This would otherwise require 32 bits.

We will describe some bits later, such as B, E, and W in the data segment descriptor and C and R in the executable segment descriptor. As mentioned in Chapter 2, code segments have a D (default operand/address size) bit that determines whether the default size for data and addresses is 16 bits (0) or 32 bits (1).

Other bits shown in Figure 5-4 either have specific values (0 or 1) or are available for use by systems programmers (AVL). For example, one bit might identify segments containing descriptor tables, key operating system functions, and memory-mapped I/O devices that must remain in physical memory.

Note, in particular, how the granularity bit works. If it is 0, the segment ends at the limit. Its size is the limit plus 1 (including the zeroth byte at the base address). References to offsets larger than the limit cause exceptions.

If the granularity bit is 1, the segment ends at the limit shifted left 12 bits with low-order 12 bits inserted. For example, suppose $G = 1$ and the limit is 10000 hex. The segment then ends at address 10000FFF hex and its size is 10,001,000 bytes. To get a segment of size 10,000,000 hex bytes (256 Mb), you would have to specify a limit of FFFF.

Thus segments can have any size up to 1 Mb. Larger segments can only have sizes in units of 4 Kb, such as 1 Mb + 4 Kb, 1 Mb + 8 Kb, etc. In practice, of course, segments are usually relatively large and have sizes with large, discrete steps anyway. Hence, this limitation is seldom significant. If you set your heart on creating a segment with a length of 100,007 bytes hex, you will be cruelly disappointed.

Segment descriptors must be placed in one of two kinds of descriptor tables (see Figure 5-5):

- Global, that is, applying to all tasks in the system.
- Local, that is, applying only to the current task. Chapter 6 describes tasking in detail.

The 32-bit linear base address of the global descriptor table is in the global descriptor table register (GDTR). The 32-bit linear base address of the current local descriptor table is in the local descriptor table register (LDTR). The enlightening names here allow for examination questions almost as profound as Groucho Marx's "Who is buried in Grant's Tomb?" Note that the base addresses are linear, not segment relative. The GDTR and LDTR also contain 16-bit limits (the offsets of their last valid bytes). The tables consist of 8-byte entries as shown in Figures 5-4 and 5-5.

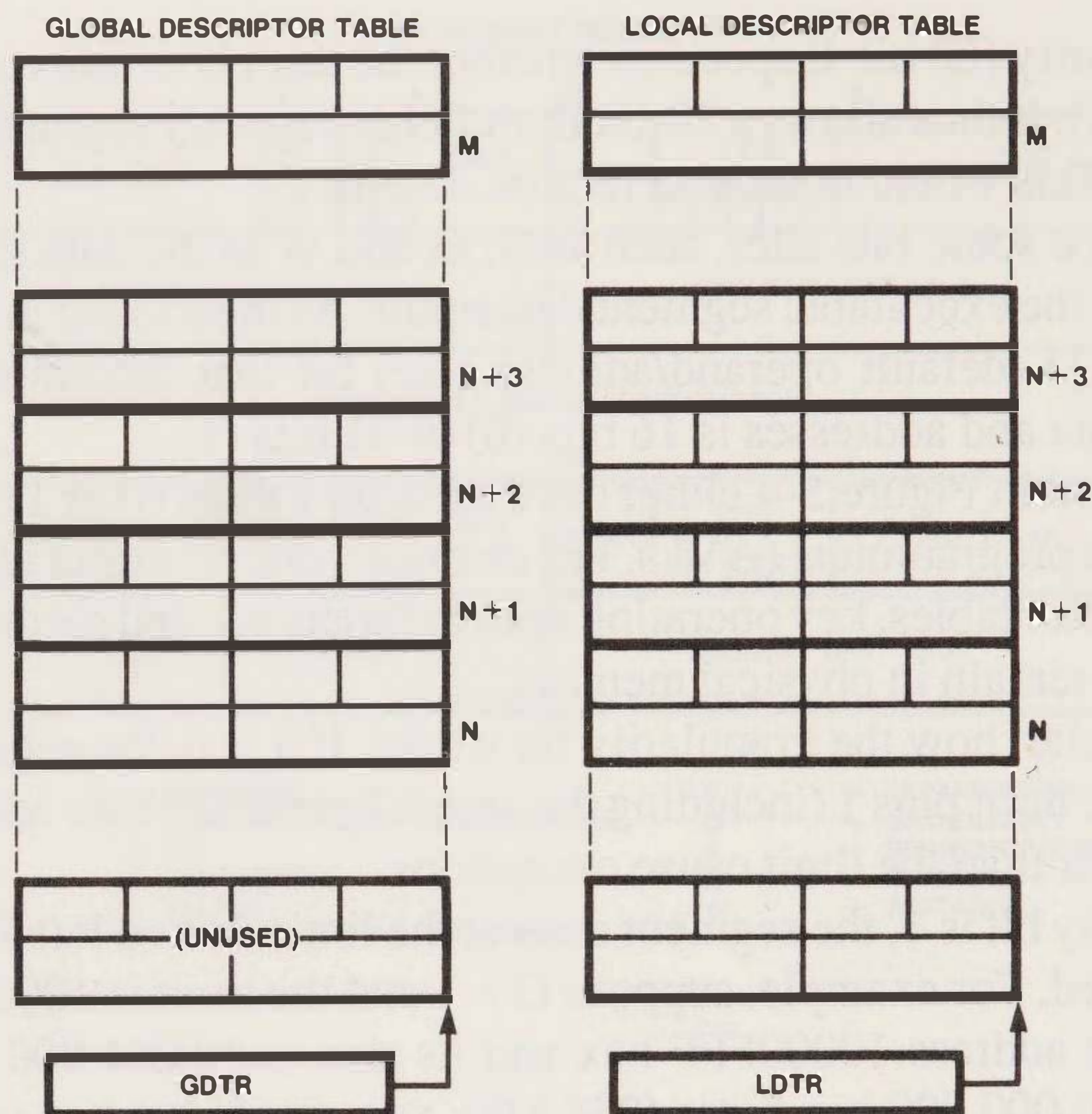


Figure 5-5
Global and local descriptor tables.

A selector thus must both specify a table and provide an index to a descriptor in it. Figure 5-6 shows the format of a selector. Since the table entries are 8 bytes long, only the high 13 bits are needed for the index anyway. Of course, the index must be multiplied by 8 before the processor uses it to access the table. Bit 2, the table indicator, determines whether the descriptor is in the GDT (0) or in the current LDT (1). Bits 0 and 1 are the requestor's privilege level (RPL); we will discuss it later in this chapter.

Some example selector values are:

1. 3005 hex. This value refers to a descriptor in the current LDT, as bit 2 = 1. The descriptor is in linear addresses $\text{LBASE} + 3000 \text{ hex}$ through $\text{LBASE} + 3007 \text{ hex}$, where LBASE is the contents of the local descriptor table register. Note that LBASE is a 32-bit linear address, not an offset.
2. AF00. This value refers to a descriptor in the GDT, as bit 2 = 0. The descriptor is in addresses $\text{GBASE} + \text{AF00 hex}$ through $\text{GBASE} + \text{AF07 hex}$, where GBASE is the

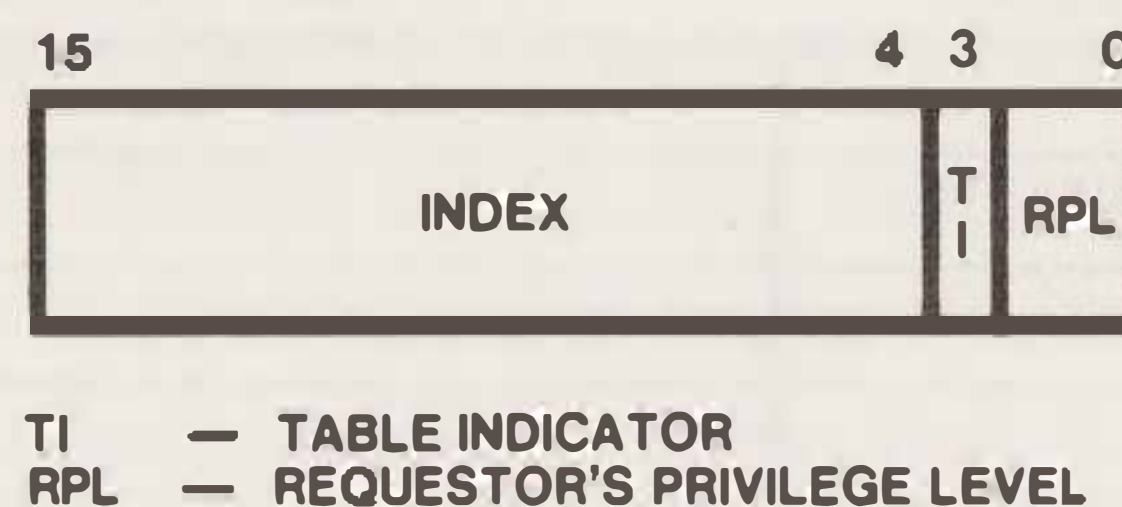


Figure 5-6
Format of a selector.

contents of the global descriptor table register. Like LBASE in the previous example, GBASE is a 32-bit linear address.

3. 0 (zero). This is the famous null selector in the GDT, as bit 2 = 0. It serves only to identify unused segment registers and other invalid segment values. Think of it as comparable to the null character (often designated as `\0`) that ends strings in languages such as C.

How does address translation work in practice? Here are some examples:

1. Suppose that an instruction gets data from address (offset) A5B370 and the data segment register contains 150 hex. This selector refers to a descriptor in the GDT as bit 2 = 0. The descriptor is in linear addresses GBASE + 150 hex through GBASE + 157 hex, where GBASE is the contents of the global descriptor table register. Suppose that the base address in the descriptor is 4F1100. The data's physical address is

$$A5B370 + 4F1100 = F4C470$$

Note that no multiplication or shifting is necessary. There is no simple arithmetic relationship between the data segment register's contents and the segment's base address. Furthermore, the translation depends on what an operating system puts in the global descriptor table.

2. Suppose that an instruction starts at address (offset) 6D11F3 hex and the code segment register contains 2004 hex. This selector refers to a descriptor in the current LDT as bit 2 = 1. The descriptor is in linear addresses LBASE + 2000 hex through LBASE + 2007 hex, where LBASE is the contents of the local descriptor table register. Suppose that the base address in the descriptor is 117E00. The instruction's physical address is

$$6D11F3 + 117E00 = 7E8FF3$$

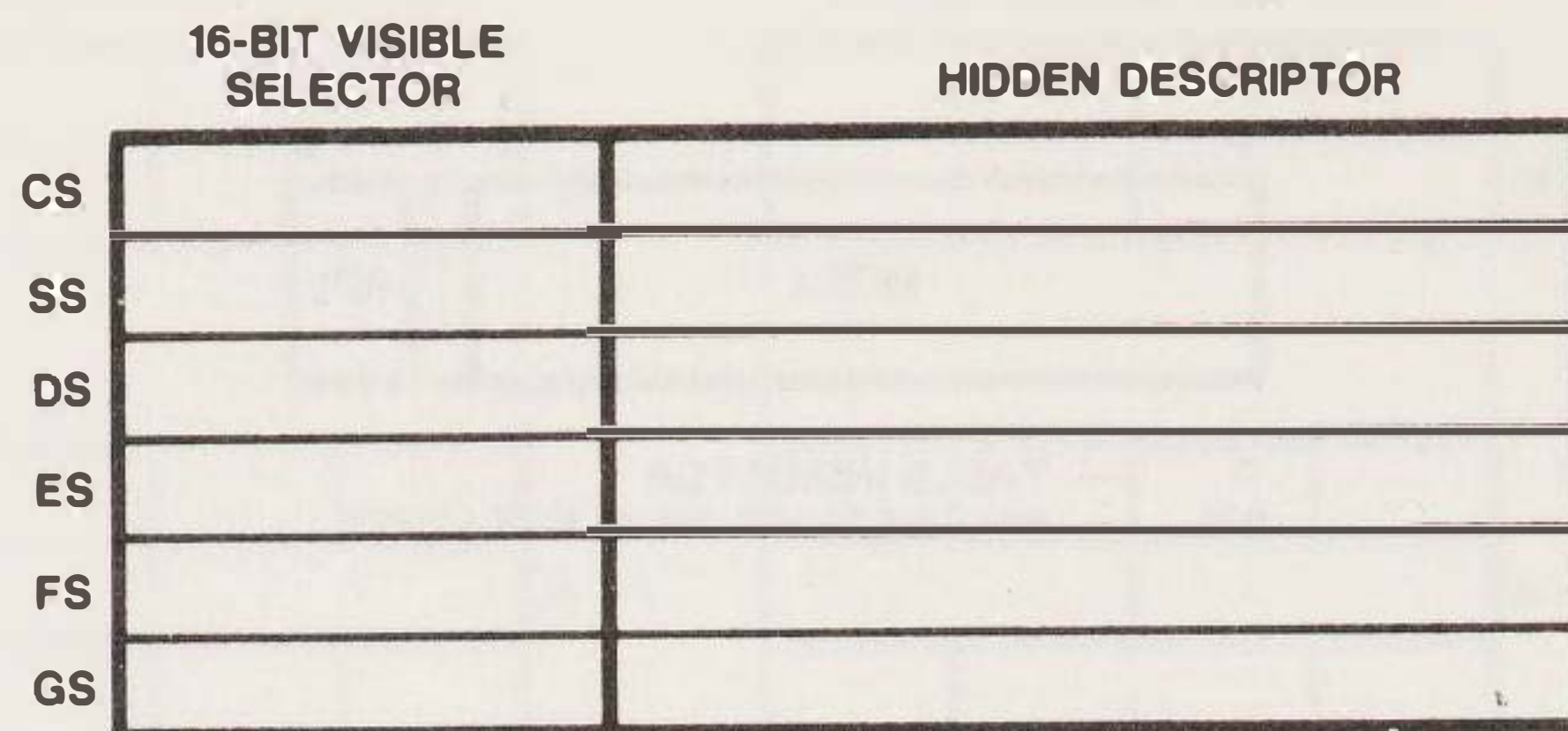


Figure 5-7
Segment registers.

Here the translation depends on which local descriptor table is currently in use.

An important fact to remember is that the processor does not use the GDT's first entry (index 0). This allows the system to load the zero (null) selector into any segment register without causing an exception. Note that zero refers to a selector in the global table, as bit 2 = 0. The zero selector is always valid for loading but never for use. Erroneous references to cleared segment registers thus cause exceptions. The zeroth entry is valid in local descriptor tables.

Selectors do not actually occupy entire segment registers. In fact, the registers (see Figure 5-7) have a visible part containing the selector and a hidden part containing the base address, limit, type, and other attributes obtained from the descriptor table. You may compare the hidden part to the attributes of a file that you select by name. The terms visible and hidden refer to the programmer's point of view. Common instructions such as MOV manipulate only the 16-bit "visible" parts. You can, however, retrieve the segment limit with the LSL (load segment limit) instruction and the access rights with the LAR (load access rights) instruction.

In general, the hidden parts are freeloaders that "come along for the ride." The processor loads one automatically when it loads the selector. Having the hidden part on chip saves memory accesses each time the processor uses the selector. That is, the processor need not read the descriptor to compute the linear address or to do validity checks. The information it needs is available in the hidden part of the segment register.

In this indirect mapping, successive selectors or ones with only slightly different values may produce completely different linear addresses. For example, remember that in the 8086 segments A000 and B000 follow each other in physical memory (assum-

ing that both are 64 Kb long) just as in logical memory. In the 80386, on the other hand, segments A000 and B000 refer to different entries in the global descriptor table. Their base addresses, limits, and attributes are totally independent. Thus, in the protected mode, there is no simple way to compute the linear address for the next higher logical address in a new segment. Furthermore, programs that assume simple mappings when handling large code or data areas or refer to physical addresses will not work in protected mode. This is one reason why MS-DOS programs often cannot run in protected mode on either the 80286 or the 80386.

PAGING

Unlike segmentation, paging is optional on the 80386. The PG bit (bit 31 of control register 0) determines whether it is activated. This bit must be 1 to activate paging. Like the choice between real and protected modes, the selection of paging is a systemwide decision. The switch to paging is a drastic step that a system would undertake just once at startup.

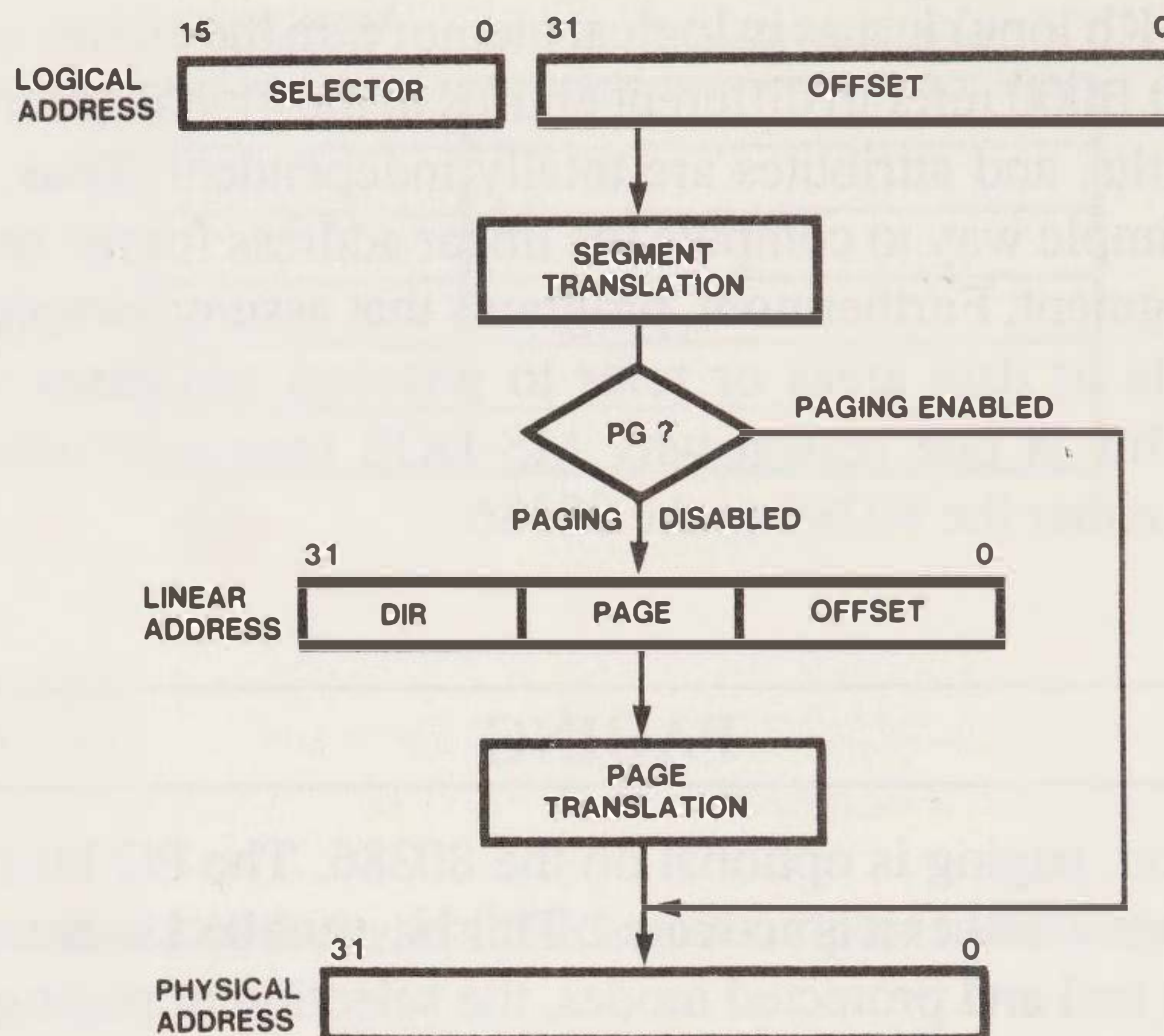
The operating system can only set the PG bit in the protected mode. It must therefore initialize PG after or at the same time as it sets the PE bit to enter protected mode. There is no paging in real mode but there can be in V86 mode. Paging in V86 mode allows several users to have their own separate “virtual” 8086-based machines. Note the obvious advantage of the more flexible mapping method here.

On the 80386, paging occurs after segmentation as shown in Figure 5-8. Segmentation converts the logical address into the intermediate linear address. Paging then converts the linear address into a physical address. Both conversions involve a table lookup.

Page Translation

Page translation involves two levels of tables. They are:

1. Page directories, which contain the physical base addresses of page tables. A page directory is like an index to a set of maps or local telephone directories. Page directories allow a system to have several page tables, thus keeping users or tasks separated.
2. Page tables, which contain the physical base addresses of pages.

**Figure 5-8**

An overview of 80386 address translation in protected mode.

A page directory is just like a page table except for the kind of entries it contains.

To compute the physical address, the processor divides the linear address into three fields as shown in Figure 5-9:

- Bits 22 through 31 are the page directory index. The processor uses it to select the base address of the page table from the page directory. The 10-bit field means that a directory can have up to 1K (1024) page table entries.
- Bits 12 through 21 are the page table index. The processor uses it to select an entry from the page table. The 10-bit field means that a page table can have up to 1024 page entries.
- Bits 0 through 11 are the offset on the page. The processor adds it to the page table entry much as it adds an offset to a segment base address during segmentation. Note that the two offsets are not the same unless the segment's base address is on a 4K page boundary.

Paging is thus a multistage process. To implement it, the processor needs:

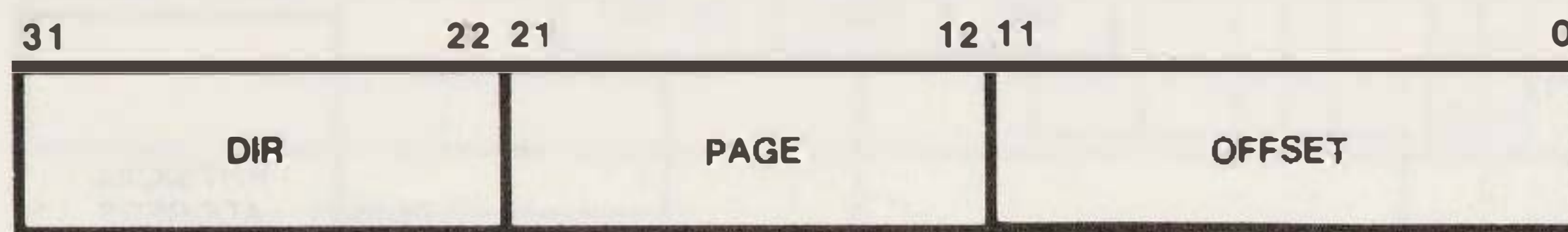


Figure 5-9

Format of a linear address.

1. The base address of the current page directory in register CR3 (also called the page directory base register). This address can be a systemwide constant or it can change on a task-by-task basis.
2. The base address of the current page table. The processor obtains it from the current page directory by indexing with bits 22 through 31 of the linear address.
3. The page table entry. The processor obtains it from the current page table by indexing with bits 12 through 21 of the linear address.

As shown in Figure 5-10, the physical address is then the sum of the offset (bits 0 through 11 of the linear address) and the page table entry. This addition is the final step in address translation.

Let us look at some examples:

1. Suppose that the linear address is 1F3A1. It consists of the following fields (see Figure 5-9):
 - The page directory index (bits 22 through 31) is 0.
 - The page table index (bits 12 through 21) is 1F.
 - The offset on the page (bits 0 through 11) is 3A1.

Paging then proceeds as follows:

- a. The processor reads the page table's base address (PTBASE) from locations DIRBASE through DIRBASE + 3. DIRBASE is the contents of the page directory base register (control register 3).
- b. The processor reads the page table entry from locations PTBASE + 7C hex through PTBASE + 7F hex. 7C is the page table index (1F) times 4, as each entry is 4 bytes long. If you feel a need to check the arithmetic here, we strongly suggest using a hex calculator. Assume that the upper 20 bits of the entry (see Figure 5-11) are PFRAME.

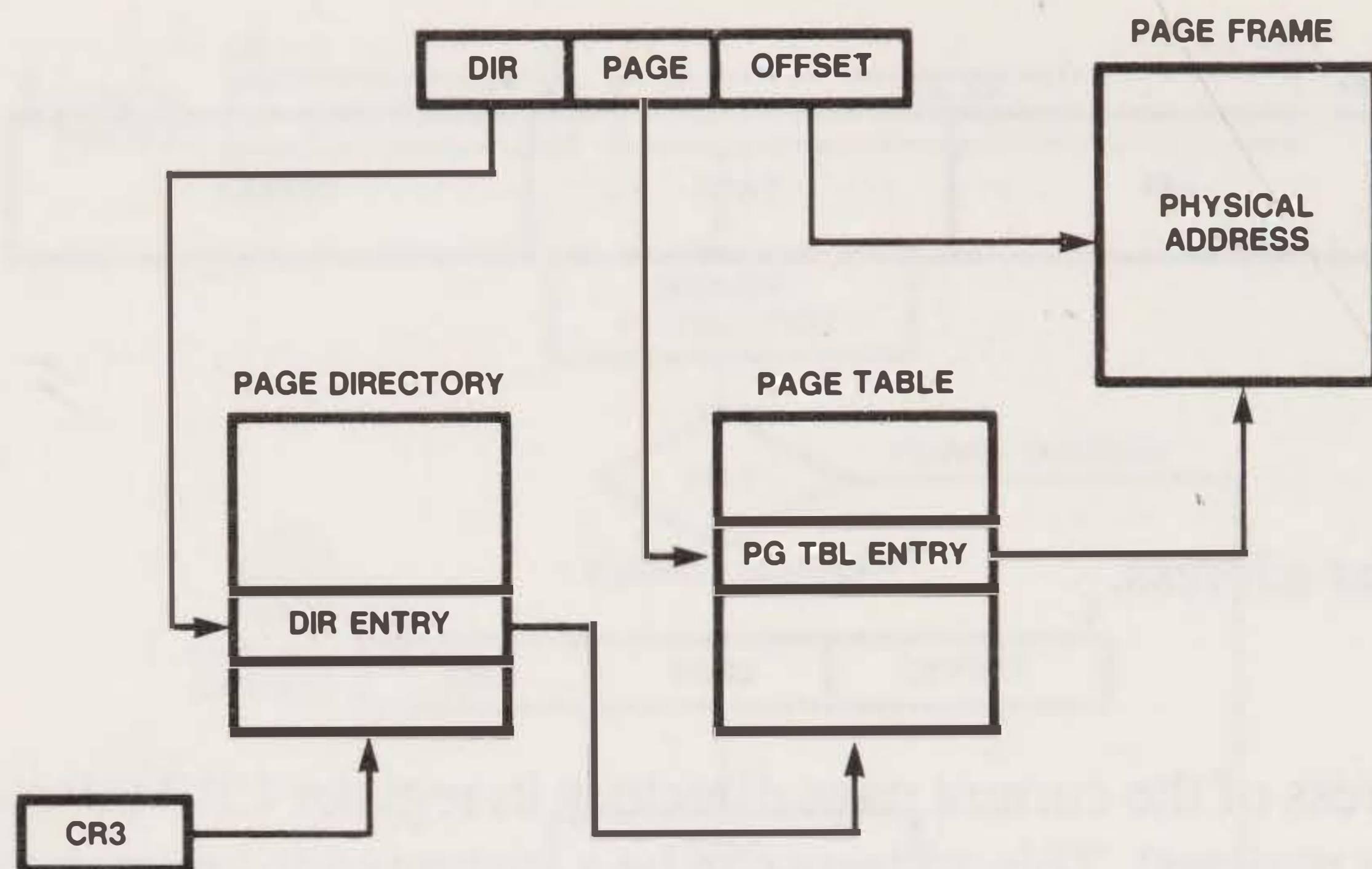


Figure 5-10

The page translation process.

- c. The processor computes the physical address by adding 3A1 to PFRAME extended with 12 low-order 0 bits. If, for example, PFRAME = 6C, the physical address is 6C3A1.
2. Suppose that the linear address is C31B79 (also a great stage name for a robot). It divides as follows:
 - The page directory index is 3 (the two upper bits of the most significant digit). Note that the break between page directory index and page table index occurs in the middle of a hex digit, since the fields are both 10 bits long.
 - The page table index is 31 hex.
 - The offset on the page is B79.

Paging then proceeds as follows:

- a. The processor reads the page table's base address (PTBASE) from locations DIRBASE + C through DIRBASE + F. DIRBASE is the contents of the page directory base register. C is the page directory index (3) times 4 in hex, as each entry is 4 bytes long.
- b. The processor reads the page table entry (PFRAME is the upper 20 bits) from locations PTBASE + C4 hex through PTBASE + C7 hex. C4 is the page table index (31 hex) times 4.

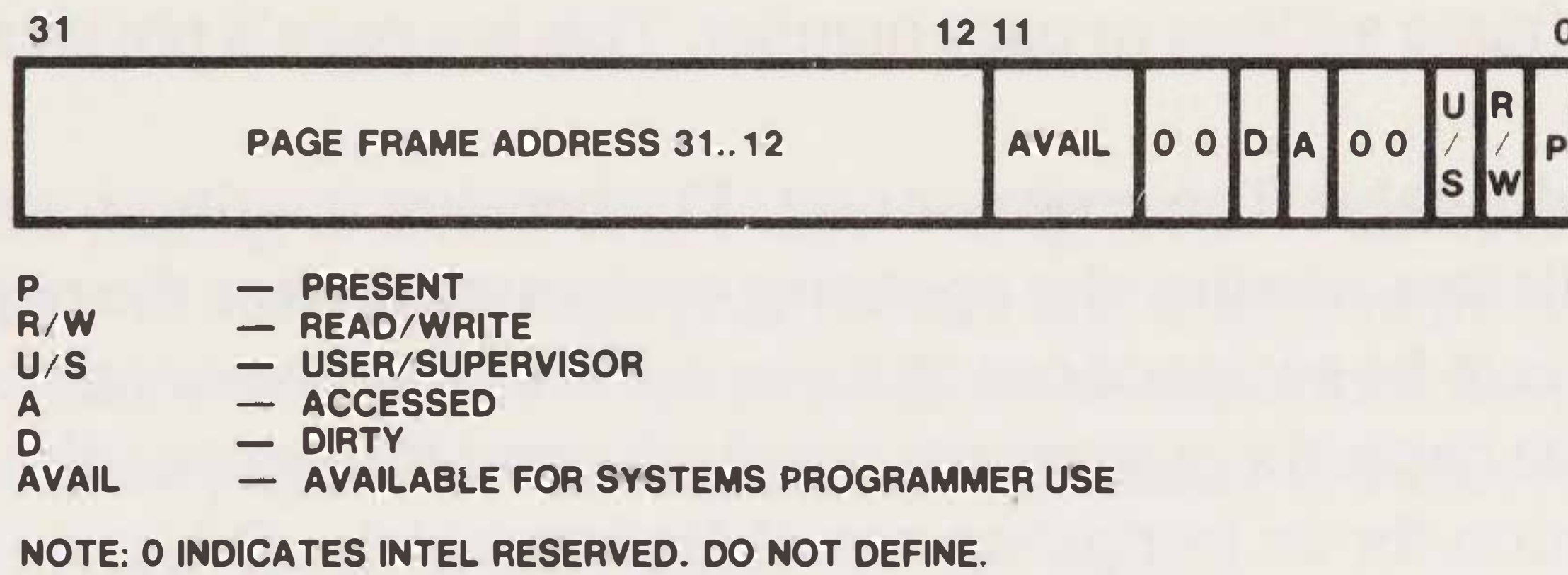


Figure 5-11

Format of a page table entry.

- c. The processor computes the physical address by adding B79 to PFRAME. is extended with 12 low-order 0 bits.

Fortunately, only operating system developers would ever have to check these calculations in practice.

Page Tables

Note the following about page tables:

1. They contain 32-bit elements consisting of the physical base address of a 4K block of memory (a *page frame*) and other attributes (see Figure 5-11). Note that page tables (including page directories) contain physical addresses, not linear addresses. Pages always start on 4K boundaries, so the low-order 12 bits of their base addresses are all zeros.
2. Each one occupies a page. It can therefore hold up to 1K entries. Thus each table can provide access to 1K pages or 4 Mb of memory. Unlike descriptor tables, page tables are always the same size. There is no hidden limit in a page directory base register or in a page directory entry. Unused page table entries should be cleared to avoid errors.
3. A page directory also occupies a page. It can therefore hold up to 1 K entries. Thus each directory can provide access to the entire physical address space, 1 K 4-Mb units or 4 Gb of memory.

Figure 5-11 shows the format of a page table entry. It has (from left-to-right) the following fields:

- A page frame address or page number. This is a page's physical base address.
- The D (dirty) bit. The processor sets D whenever it writes into a page. D thus indicates whether the operating system must save the page on disk when it is to be swapped out of memory. If the processor has never written on the page, the system can simply discard it. A rather disappointing explanation for an intriguing name! Unfortunately, D bits do not apply to politicians, White House staff members, Wall Street financiers, or football players.

Note also that a page table entry's D (dirty) bit is different from an executable segment descriptor's D (default address/operand size) bit. Only pages have dirty bits, not segments. However, a data segment descriptor has a bit indicating whether the segment is writable.

- The A (accessed) bit. The processor sets A whenever it accesses a page. The operating system can use the A bit to identify pages that have not been used lately and are therefore likely candidates for swapping.
- The U/S (user/supervisor) bit. This is a protection bit (to be discussed later) that determines whether the page is accessible by programs running at the user privilege level or only at the supervisor level.
- The R/W (read/write) bit. This bit determines whether the page is read-only (0) or read/write (1) at the user level. All pages are always readable and writable at the supervisor level. Note, however, that segment-level attributes apply first and can override page-level permissions.
- The P (present) bit. P is 1 if the page or page table is in physical memory. The processor signals a page exception or page fault if it tries to use an entry for which $P = 0$.

Note that bits 9, 10, and 11 are available for the systems programmer to use. These bits could indicate whether a particular page can be removed from physical memory. Note that the current page directory and page tables must always stay in memory. So must memory-mapped I/O devices, basic operating system functions (the kernel), the global descriptor table, and basic interrupt and exception servicing functions. You may want to keep all interrupt handlers in memory to reduce startup time (*latency*).

The system programmer may also want to indicate whether a page contains code or data. This is not inherent in page typing as it is with segments.

Page Cache

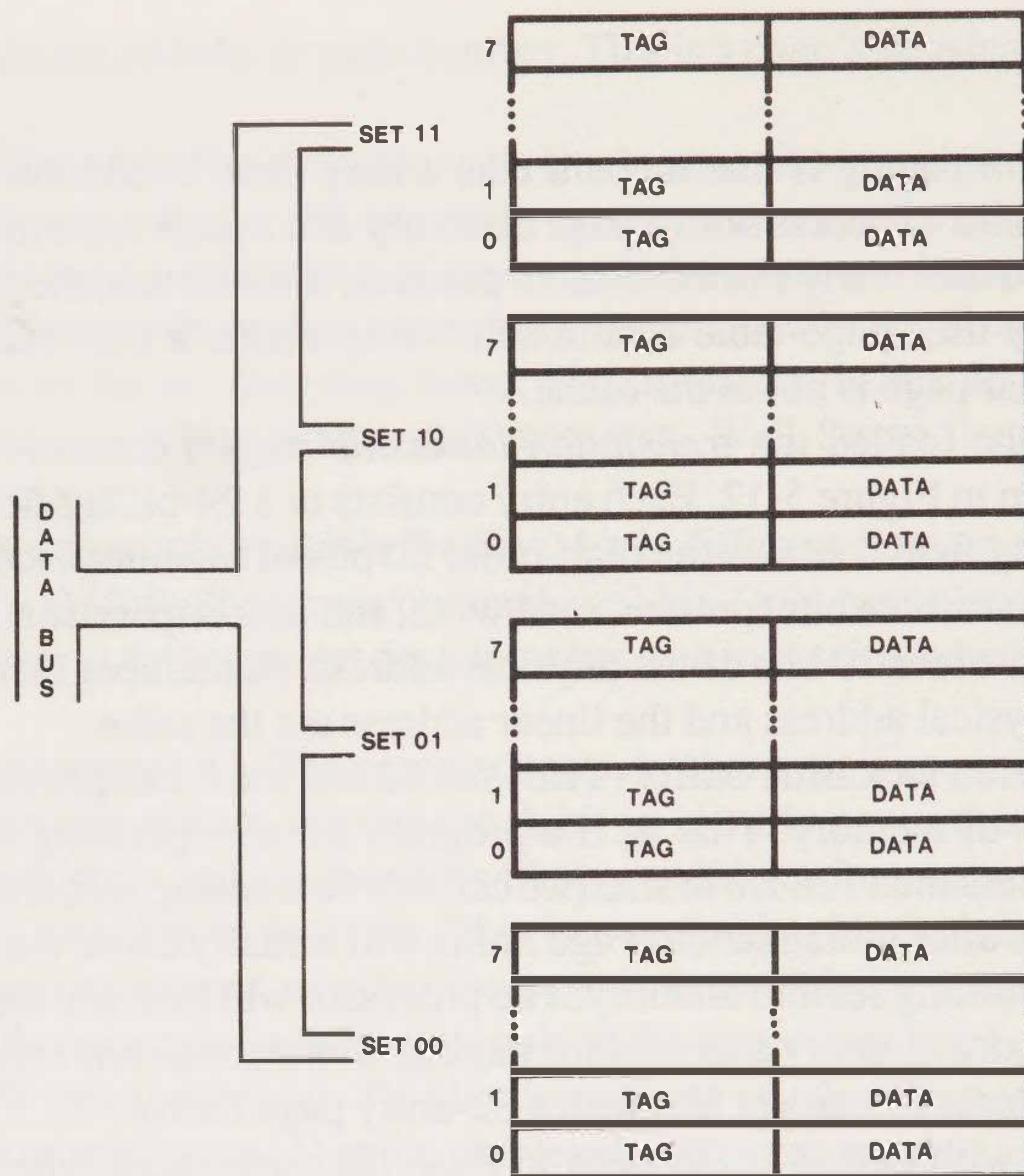
One problem with paging is that it could take a long time. If address translation required the processor to access both a page directory and a page table in memory, each instruction would take many extra cycles. In practice, to avoid this, the processor saves the most recently used page-table data in an on-chip cache. It then accesses memory only if a particular page is not in the cache.

The page cache (called the *translation lookaside buffer*) contains 32 entries organized as shown in Figure 5-12. Each entry consists of a 24-bit tag field and a 20-bit data field. The tag field contains the high-order 20 bits of the linear address, the valid bit, and the three attribute bits (present, read/write, and user/supervisor). The data field contains the high-order 20 bits of the physical address. Remember that the low-order 12 bits of the physical address and the linear address are the same.

As the translation lookaside buffer (TLB) has 32 entries, it can provide access to 32 pages or 128 Kb of memory. That is, if a program's main operating section uses an area of memory less than 128 Kb in size (we call this its *working set*), it can run without any cache misses after initialization. Page faults will initially cause the operating system to load the working set into memory. The processor will then use the set continually without any extra memory accesses for paging. Simulations have shown that most programs get at least 95 percent hits with a 32-entry page cache.

One problem with paging is the need to flush the page cache whenever the page tables change. Obviously, all the entries are now invalid, as the mapping has changed. However, the processor does not flush the cache automatically. A program must flush it explicitly either by reloading CR3 (the page directory base register) or by invoking a new task that reloads CR3.

The page cache (TLB) introduces complications. Misses make access time variable because of the unpredictable effects of interrupts, exceptions, and task switches (see Chapters 6 and 7). Furthermore, some misses occur even for entries that are in the cache. The special case occurs when the processor writes into a page for the first time. That is, the D bit in the page's entry is cleared. The processor must then go through the miss procedure just to set the D bit. Note that the procedure also sets the A bit automatically.

**Figure 5-12**

Structure of the translation lookaside buffer.

MEMORY PROTECTION

The 80386's protection features are intended to increase software reliability and provide a sound base for multitasking and multiuser systems. They help identify errors that could change memory locations improperly, conflict with other uses of system functions or I/O, or override operating systems programs. These features are particularly important in multitasking and multiuser applications. There, one program may inadvertently affect other tasks or other users as well as itself. Protection features act much like the regulations, restrictions, and penalties that help protect shared public facilities such as parks, schools, and streets.

There are five aspects to protection in the 80386:

- Type checking. An example is determining whether a segment that is going to be used for instructions actually contains code.
- Limit checking. An example is determining whether a reference falls within the limit of a segment or the bounds of an array.
- Restriction of addressable domain. An example is determining whether a segment is accessible by its caller or is protected from it.
- Restriction of procedure entry points. This forces entries into an operating system or other shared software to pass through well-defined checkpoints.
- Restriction of instruction set. This means that only trusted procedures can execute instructions that perform memory management tasks, overall system control, or input/output.

All checks occur during address generation, so they do not slow the processor.

A key element in the 80386's protection mechanisms is the concept of privilege level. 80386 descriptors can have four privilege levels (0 through 3, where 0 is the highest). These levels restrict access to a descriptor. Only callers with a privilege level at or higher than the descriptor's privilege level (DPL) can access it. An attempt by a caller at a lower privilege level causes a general protection exception (see Chapter 7).

Be careful of the fact that levels become more privileged as their numbers decrease. We will speak of "more privileged" segments and "higher privilege levels" without specifying numbers. In fact, more privileged segments are at lower-numbered levels. Fortunately, operating systems usually handle the details of privilege levels without forcing users to think in reverse.

The approach is the same as the common practice of referring to a "first team" or "first string" or to "first-rate." Thus, second or third stringers play behind first stringers, and second-rate products are inferior to first-rate ones. Also in many sports high numbers indicate players who seldom appear or are unlikely to make the team.

Systems need not use all four privilege levels. In fact, a system need not even use privilege — in this case, all its segments should be at level 0. A common situation is for a system to have user and supervisor (operating system) levels. User segments should then be at level 3 and operating system segments at level 0. Note that these privilege levels apply to segments (logical memory), not to pages (physical memory).

Pages, as we noted earlier, have their own privilege levels defined by the user/supervisor (U/S) bit. Here there are only two levels: 0 (supervisor) and 1 (user). The supervisor level, confusingly enough, corresponds to descriptor privilege levels 0, 1, or

2. The user level corresponds to descriptor privilege level 3. This makes at least as much sense as the playoff systems in major sports leagues. The end result is that:

- Procedures running at levels 0, 1, or 2 can access all pages.
- Procedures running at level 3 can access only user-level pages.

Note also that the read/write bit limits the write access of procedures running at level 3 but not of those running at levels 0, 1, or 2. Be careful of the fact that typical two-level (user/supervisor) systems use segment privilege levels 0 and 3 but page privilege levels 0 and 1.

In most cases, page-protecting memory is redundant if you have already segment protected the underlying programs and data. The 80386 checks segment privilege levels before page privilege levels anyway. However, the page protection costs nothing and may give the system designer extra piece of mind. Page protection can also help trap references to code pages or unallocated pages. Another use for it is to protect only part of a large code or data segment. This is particularly important for programs that lie entirely within a single segment.

Descriptors contain type and limit information that systems can use for protection. Note, for example (see Figure 5-4), that a data segment descriptor has a writable (W) bit that specifies whether instructions can write into it. Similarly, an executable-segment descriptor has a readable (R) bit that specifies whether instructions can read from it. A page's read/write bit can have a similar effect on user-level procedures.

The 80386 recognizes the following as protection exceptions:

- Violating segment or page privilege level restrictions.
- Loading the CS register with a selector of a nonexecutable segment.
- Loading any data segment register with a selector of an unreadable executable segment.
- Loading the stack segment register with a selector of a nonwritable segment.
- Trying to write into an executable segment. In protected mode, the 80386 thus does much more than just politely discourage self-modifying code. An interpreter or debugger can, however, write into a code segment by defining a contiguous data segment. That is, you must define two separate segments that refer to the same addresses.
- Trying to write into a data segment or a page that is not writable.
- Trying to read from an executable segment that is not readable.

The processor uses a segment descriptor's limit field to check references. Only ones that fall within the limit are valid. A complicating factor in this determination is the E bit in a data segment descriptor. This bit, the expansion-direction bit, decides whether

Table 5-2

Useful combinations of the E (expansion-direction), G (granularity), and B (big) bits in data segment descriptors

Case:	1	2	3	4
Expansion Direction	U	U	D	D
G-bit	0	1	0	1
B-bit	X	X	0	1
Lower bound is: 0 LIMIT + 1 shl(LIMIT, 12, 1) + 1	X	X	X	X
Upper bound is: LIMIT shl(LIMIT, 12, 1) 64K - 1 4G - 1	X	X	X	X
Max seg size is: 64K 64K - 1 4G - 4K 4G	X	X	X	X
Min seg size is: 0 4K	X	X	X	X

shl (X, 12, 1) = shift X left by 12 bits inserting one-bits on the right

the segment is the usual (expand up) or expands down. In the expand-down case (used mainly for stacks), the range of valid addresses is from limit + 1 to either 64K or 4 Gb, depending on the B (big) bit. The upper limit is 64K if the B bit is 0 and 4 Gb if it is 1.

Note the following special features of an expand-down segment:

- It has maximum size when the limit is zero.
- You can expand a stack in size by copying it to a larger segment without updating intrastack pointers. The stack then simply occupies the upper part of a larger area.
- Table 5-2 shows useful combinations of the E, G, and B bits.

Fortunately, expand-down segments are rare in practice. The only exceptions are stacks that the operating system usually manages automatically anyway.

Domain Restrictions

When the processor loads a data segment's selector into a data or stack segment register (DS, ES, FS, GS, or SS), it automatically evaluates access from the currently executing segment. Figure 5-13 shows how this works. The evaluation involves comparisons of three privilege levels:

- The CPL (current privilege level) of the executing segment
- The RPL (requestor's privilege level) of the selector
- The DPL (descriptor privilege level) of the target segment's descriptor

An instruction may use the target segment only if its DPL is larger than or equal to both the CPL and the selector's RPL. Remember that less privileged segments have higher privilege numbers. Thus the idea is that the target segment must be equal or lower in privilege than either the CPL or the RPL.

Suppose, for example, that the CPL is 3. That is, the current privilege level is the lowest (user level). User programs can access only selectors and descriptors at level 3.

A special situation is one in which a routine running at privilege level 3 tries to pass an operating system routine (running at privilege level 0) a selector with privilege level 0. The operating system routine would then be able to use the selector, as it has the appropriate privilege level. The way to foil this backdoor effort is to use the ARPL (Adjust Selector's RPL Field) instruction to reduce the selector's privilege level. ARPL adjusts the selector's RPL to not less than the caller's CPL. The descriptor (at privilege level 0) is now inaccessible because it is more privileged than the selector.

The idea behind this kind of restriction is to prevent user programs from changing page directories, page tables, descriptor tables, or values internal to the operating system. For example, user programs could not change disk parameters or interrupt-handling routines as they can in current versions of MS-DOS.

Restricting Control Transfers

The introduction of privilege levels raises several new questions such as:

How can a user routine call an operating system function (or *utility*) that it may need?

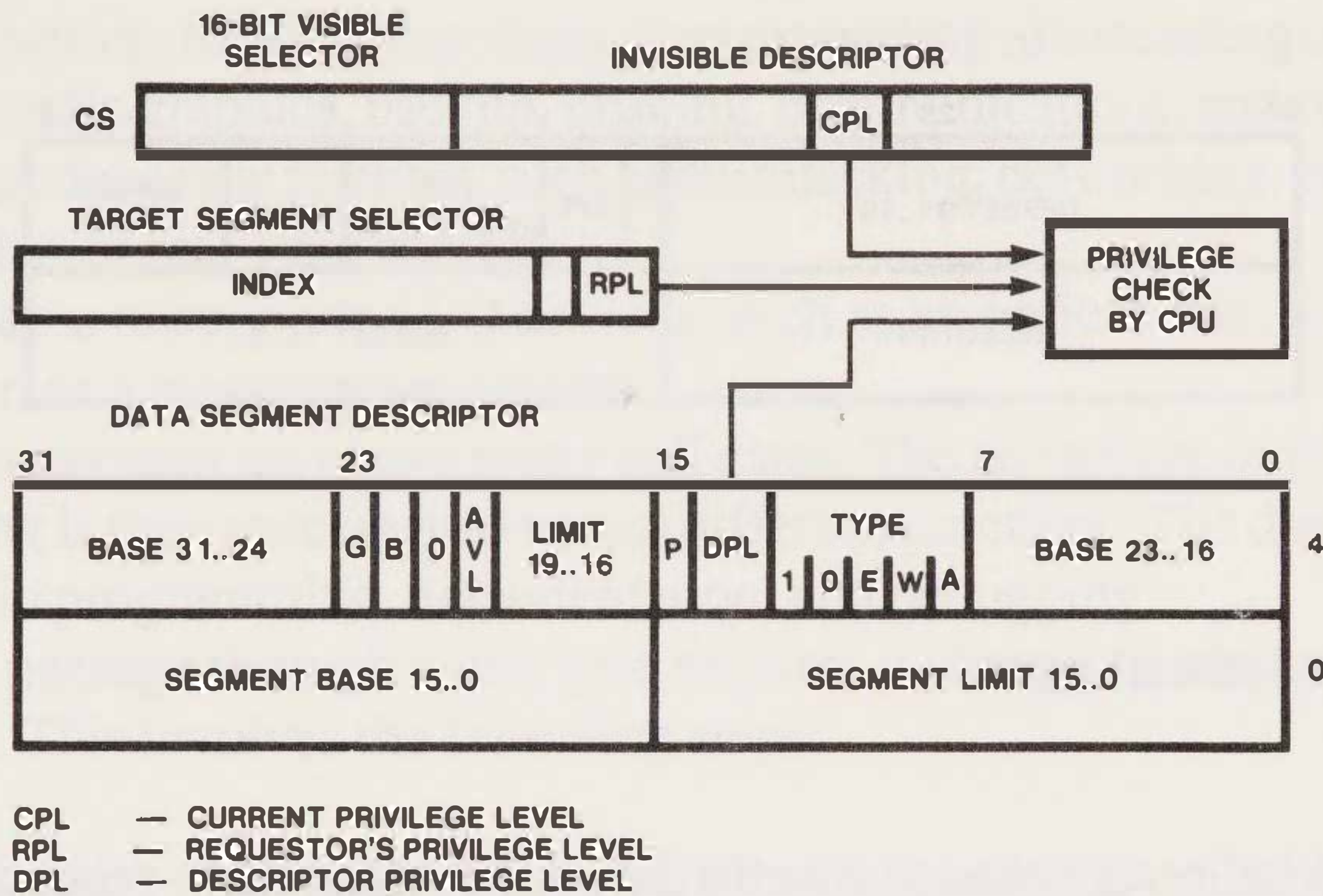


Figure 5-13

Privilege check for access to a data segment.

How do you write general functions such as code conversions, mathematical routines, and utilities that programs at all privilege levels can access?

The answer to the first question is to use gate descriptors. There are four kinds:

- Call gates
- Trap gates
- Interrupt gates
- Task gates

We will describe only call gates here. We will discuss task gates in Chapter 6 and trap and interrupt gates in Chapter 7.

Gates reside in either the global descriptor table or a local descriptor table. Call gates usually are in the global descriptor table so that all tasks have access to them.

We can think of a call gate as a border crossing station. It allows only legal entries. Here is where the authorities look at your papers and determine whether you can pass to the other side. Similarly, call gates both define a procedure's entry point and specify its privilege level. The hardware recognizes references to call gates and expands CALL instructions appropriately.

Figure 5-14 shows the format of a call gate. It has selector and offset fields that form a pointer to the entry point. Thus a call gate basically provides an indirect transfer to a system procedure. The selector must refer to the descriptor of an executable segment.

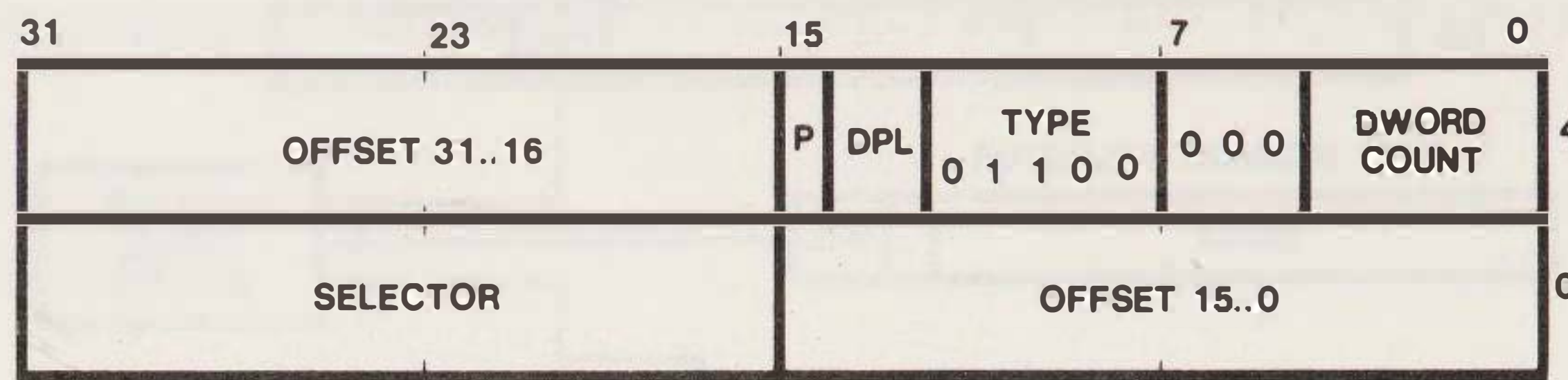


Figure 5-14
Format of an 80386 call gate.

The calling instruction only has to specify the gate's selector; the processor ignores the offset. A call gate also has a dword count that indicates how many double words the processor must move from the caller's stack to the new privilege level's stack. Note that each privilege level has its own stack to maintain system integrity.

Call gates guarantee legitimate entry points. There are no backdoors or side entries into the operating system as there are in MS-DOS. The idea is to force all software to be well-behaved and therefore capable of running together with other programs and under new versions of the operating system. MS-DOS programs are often incompatible and incapable of running under new DOS versions because they use nonstandard entry points and interfere with DOS' internal workings. Note, for example, the well-known inability of many memory-resident programs (such as Borland's SideKick and RoseSoft's ProKey) to work together or at the same time as other programs such as Microsoft Word and Lotus 1-2-3.

Why do programs often skirt or avoid restrictions imposed by an operating system? Typical reasons are:

- To do operations (such as screen updates and disk transfers) faster than the OS supports by accessing memory or I/O directly.
- To control I/O operations (such as keyboard entry) in more detail than the OS allows. For example, the program may want to use key combinations that the OS does not recognize or to differentiate between key closures and releases.
- To provide immediate activation of features by intercepting OS calls. Seldom-used key combinations can then provide direct access to a resident program.

- To substitute for DOS functions, thus expanding or extending commands that handle graphics, backup, printing, communications, and other tasks.
- To introduce new features, such as multitasking, networking, encryption, security, or real-time control.
- To access unsupported I/O devices, such as an optical disk, a document reader, or a mass storage system.

An operating system may have many call gates. The advantage of more gates is that less dispatching is then necessary to access different functions. The disadvantage is the complication in programming, documentation, and debugging.

Successful passage through a call gate requires a change of stacks if the privilege level changes. This involves the following steps:

1. Checking the size of the new stack. If it is not large enough to hold the parameters and linkages, a stack fault occurs (see Chapter 7).
2. Pushing the old values of the SS and ESP registers onto the new stack. They provide the linkage back to the previous stack. The transfer of SS is a 32-bit operation in which the upper 16 bits are wasted.
3. Copying parameters from the old stack to the new stack. The double word count is in bits 0 through 4 of the first double word of the call gate (see Figure 5-13). The count may be as large as 31; it does not include parameters passed in registers. The processor does not perform any validity checks on the parameters.
4. Pushing a pointer to the instruction after the CALL onto the new stack. This pointer consists of a code segment selector and an instruction pointer value. It provides a link back to the calling program.

The new stack therefore contains the following items, starting from the top (see Figure 5-15):

- Instruction pointer value for the instruction after the CALL
- Code segment register value for the instruction after the CALL (extended to 32 bits)
- Parameters copied from the old stack
- Old stack pointer value
- Old stack segment register value (extended to 32 bits)

The OS can use the old stack pointer and stack segment register values to copy more parameters if necessary.

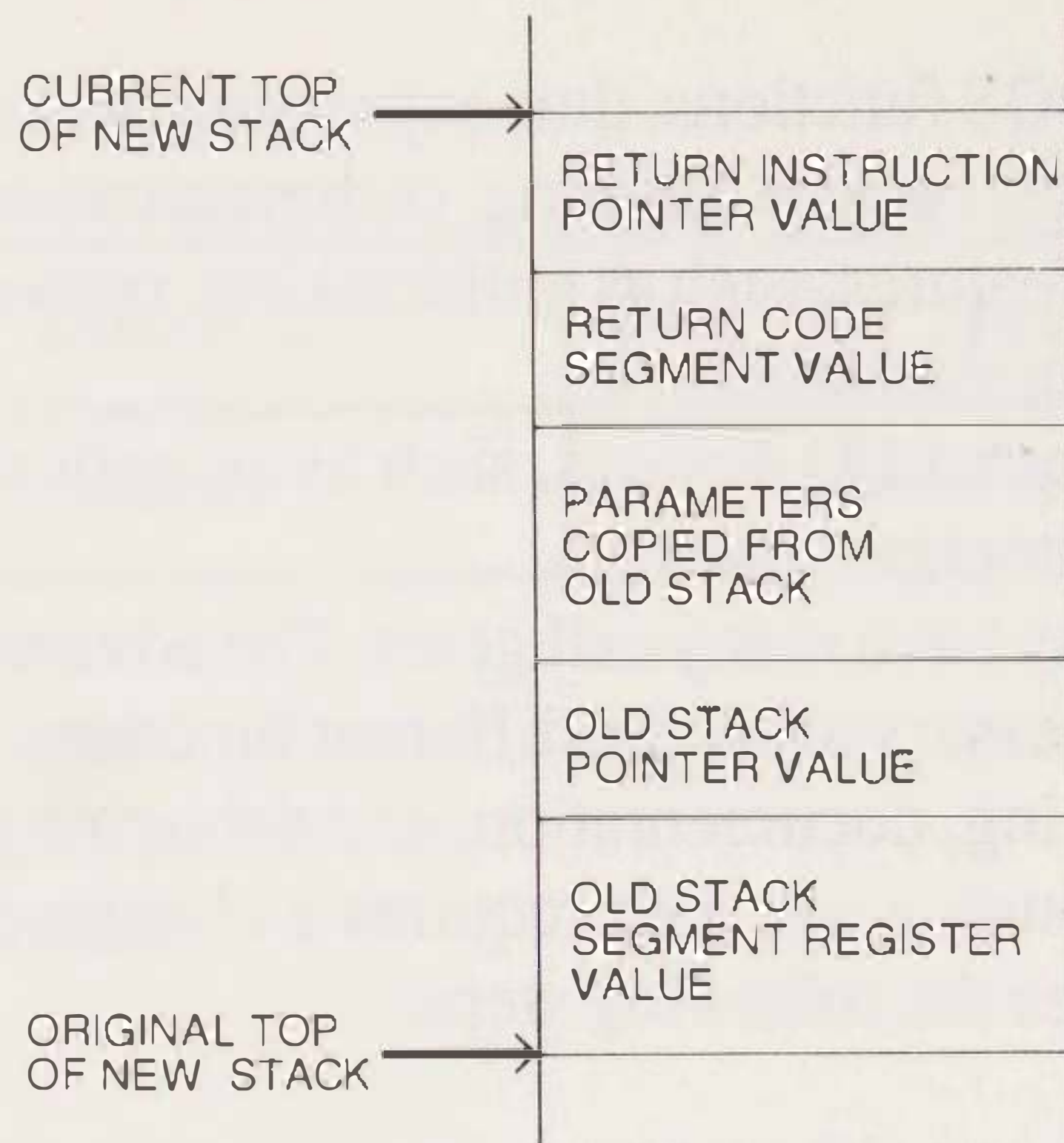


Figure 5-15

New stack contents after a change of privilege levels.

The parameter count is a fixed value for a given call gate. Copying parameters is thus awkward when one call gate provides access to several systems routines. Separate gates, each with its own parameter count, are a better approach.

Of course, what goes up (in privilege level) must eventually come down. The opposite of the CALL through a call gate is a return (RET) instruction. RET can change privilege levels, but only downward. The reason for this restriction is that a sneaky underprivileged program could otherwise use an RET to access a higher level. After all, who would know that no one had ever called it in the first place? It's like returning innocently after intermission to a performance for which you had no ticket.

However, a more privileged program can always call a less privileged program by pushing its address onto the stack and doing an RET. This is like joining the returning crowd after you missed the first part of a performance. It may seem sneaky, but you haven't violated any rules. Besides, you end up with a great alibi for a murder mystery. "I can assure you that Lord Peter Witless was at the symphony that evening. I distinctly remember him snoring quite loudly through the entire second half of the performance."

An RET that returns to a less privileged level works as follows:

1. It does segment checks and then loads the instruction pointer, code segment register, stack pointer, and stack segment register from the stack.
2. It adjusts the old stack pointer by a number of bytes given as a parameter.

3. It checks all data segment registers (DS, ES, FS, and GS) and clears any that refer to segments more privileged than the new privilege level. This prevents the new code from accessing more privileged segments using leftover selectors. The situation is like a social climber who makes contacts using an aristocrat's discarded stationery.

Note that the processor need not save the current stack pointer. Presumably its value has not changed.

Conforming Code Segments

The 80386 also provides a way to create procedures that can run at any privilege level. These could include mathematical functions, code conversions, and other general-purpose utilities. The method involves setting the C (conforming) bit in the type information of a code segment descriptor (see Figure 5-4).

When the processor transfers control to a conforming segment, it does not change the current privilege level. The segment thus executes at whatever privilege level its caller has. This is the only case in which the current privilege level may not be the same as the descriptor privilege level for the current executable segment. A program can do a JMP or a CALL to a conforming segment; it need not go through a gate, regardless of the segment's inherent descriptor privilege level. Exception handlers can also be conforming segments. This is convenient for divide faults, overflow, and array bounds checks in which the handler need only access the current task's data. It does not have to access operating system functions as it would to deal with page or segment faults.

CREATING DESCRIPTORS

Obviously, creating descriptors is a complex job. There are several different formats, each with many fields. In common practice, only compilers and operating systems create descriptors and arrange them in tables. Anyone who needs to do this will generally use special software tools. They let you define tables and descriptors in simple terms through a special language. The tool then produces the properly formatted output.

For example, Intel offers a 386 System Builder (BLD386) that runs under XENIX. It consists of a binder, a builder, a librarian, and a mapper. No candlestick maker, but they do form a fine barbershop quartet. The binder links modules and creates a loadable

form. The builder is the actual working program. The librarian allows you to maintain and manage library files for use in building new applications. The mapper generates printed information such as segment maps, gate maps, and symbol maps from object files.

PRIVILEGED INSTRUCTIONS

Still another aspect of privilege is the existence of instructions that only privileged procedures can execute. We will discuss system control instructions here and I/O-related instructions in Chapter 6. A procedure can execute the following instructions only if its current privilege level is 0:

CLTS	Clear Task-Switched Flag
HLT	Halt Processor
LGDT	Load Global Descriptor Table Register
LIDT	Load Interrupt Descriptor Table Register
LLDT	Load Local Descriptor Table Register
LMSW	Load Machine Status Word
LTR	Load Task Register
MOV to/from CRn	Move to Control Register n
MOV to/from DRn	Move to Debug Register n
MOV to/from TRn	Move to Test Register n

These instructions initialize many of the memory management system's pointers and parameters, such as the global and local descriptor table registers and the page directory base register (control register 3). Obviously, these instructions would not appear in most user programs anyway.

INITIALIZATION OF MEMORY MANAGEMENT SYSTEMS

After RESET, the processor starts in real mode at physical address FFFFFFFF0H. The first far (intersegment) JMP or CALL makes the processor continue in the lowest 1 Mb of physical memory. The processor must then:

- Set the PE flag to enter the protected mode.
- Initialize the global descriptor table and the GDT register. Initialize local descriptor tables and the LDT register if necessary.

- Initialize the page directories, page tables, and page directory base register if the system is using paging. Also set the PG bit to 1.

The processor can do most of these actions either in the real mode or in the protected mode. The simpler approach is to do them in the real mode. Then you need not worry about implicitly using protected mode features (such as the global descriptor table) that you have not yet initialized. Be sure to do the following:

- Put a JMP immediately after the instruction that sets the PE flag. This clears the instruction prefetch queue, eliminating information related to the real mode.
- Set PG after setting PE or at the same time. You cannot set PG when the processor is in the real mode.
- Put a JMP immediately after the setting of the PG flag. This ensures consistent addressing before and after the enabling of paging.

SUMMARY

The 80386 can operate in two modes: real mode and protected (virtual) mode. In real mode, it acts like a fast 32-bit version of the 8086 processor. In protected mode, it acts like a 32-bit version of the 80286 processor. Most applications use real mode only for initialization and use protected mode for actual operations. Within protected mode, the 80386 can operate in virtual 8086 mode. This allows it to run 8086 (particularly MS-DOS) software simultaneously with software that uses new 80386 features.

The 80386 divides program, data, and stack areas into units called segments. It provides access to up to six segments at a time through segment registers. Addresses within a segment are called offsets.

In its 8086-like modes, the 80386 uses 16-bit offsets. The segment registers contain the base addresses of segments divided by 16. The processor computes a physical address by multiplying a segment register's contents by 16 and adding the offset. The relationship between logical and physical addresses is thus a simple arithmetic function.

In protected mode, the 80386 uses 32-bit offsets. The segment registers contain indexes into tables of descriptors. Each descriptor, in turn, contains a segment's base address, limit, and other attributes. The processor computes intermediate results (*linear addresses*) by obtaining the base address from a descriptor table and adding the offset. Descriptors may be in either a global descriptor table that applies to all tasks or local descriptor tables that apply only to particular tasks.

In protected mode, the 80386 can also implement paging. Pages are 4-Kb units of physical storage. The processor accesses them through a two-stage process involving a page directory and page tables. The 80386 automatically keeps information on recently accessed pages in a special on-chip cache to avoid repetitive memory operations. Attempts to access pages that are not currently in memory cause page-fault exceptions. The operating system must then read the page from disk.

The 80386 offers many protection mechanisms for preventing improper accesses. These include:

- Four levels of privilege for descriptors and two levels for pages
- Segment and page attributes that determine type and accessibility
- Segment bounds included in the descriptors
- Call gates that restrict access points to privileged routines
- Special status, control, and memory management instructions that only privileged programs can execute.

References

Deitel, H. M. *An Introduction to Operating Systems*. Reading, MA: Addison-Wesley, 1984.

Fuhrt, B., and V. Milutinovic. "A Survey of Microprocessor Architectures for Memory Management," *IEEE Computer*, March 1987, pp. 48–67.

Tanenbaum, A. S. *Operating Systems: Design and Implementation*. Englewood Cliffs, NJ: Prentice-Hall, 1987. This book contains the complete C code listing of a Unix-compatible, but noncommercially restricted, operating system.

80386 Task Management

*Delightful task! To rear the tender thought,
To teach the young idea how to shoot.*

J. Thomson, *The Seasons*. Spring

*Do not pray for easy lives. Pray to be stronger
men! Do not pray for tasks equal to your
powers. Pray for powers equal to your tasks.*

P. Brooks, *Going Up to Jerusalem*

This chapter describes 80386 task management techniques. It first explains the reasons behind tasking and then discusses 80386 tasking features, task switching, task linking, address spaces, I/O privilege levels, I/O permission bit maps, and the initialization of tasking systems. Tasking is a key to understanding the 80386 because its memory management, protection, and exception handling facilities are all task based.

WHAT IS TASKING?

A task is just a program together with its associated data and assigned memory areas. It is like a process in Unix and other multiprogramming or multiuser operating systems. In most systems, a task is self-contained. It is a sovereign body, with its own entry points, program code, data areas, stack, and current state or status (often called its *context*). A task may have subroutines, data structures, and message passing techniques. It may call library programs and use other shared facilities.

For example, let us consider an accounting system. It may have tasks that do the following:

- General ledger
- Accounts payable
- Accounts receivable
- Payroll
- Report writing
- Audit preparation

Generally, the user will select one of these tasks from a menu. They are typically completely independent programs linked through common data structures or a database.

These tasks may, in turn, consist of subtasks. The general ledger task may, for example, include the following subtasks:

- Data entry
- Account file management
- Sorting
- Query
- Printing
- End of period processing

Each subtask is, in turn, a self-contained entity.

Similarly, we may describe an energy management system as consisting of tasks. Its tasks might be

- System setup
- Operator interaction
- Data monitoring
- System control
- Alarm recognition
- Status reporting
- Time-keeping
- Emergency handling

Some of these, such as operator interaction and alarm recognition, may work via interrupts. That is, once set up, the management system simply monitors its inputs and controls heating and air conditioning in a continuous loop. The operator has a special button that suspends normal operations and activates either a manual override or a task that accepts commands. Alarms similarly force immediate action.

Note that the priority of the tasks is critical here. The operator and alarm tasks must take priority over normal operations. Similarly, status requests must be able to interrupt the usual monitoring and control. The overall system must be able to stop one task, start another, and then resume the first task where it left off. It is thus important to be able to:

- Start and stop tasks.
- Suspend a task and resume it later. It makes no sense, for example, to have a printing task remain in control when the printer is busy, offline, or malfunctioning.
- Pass information from one task to another. For example, the alarm task must be able to provide a description of what happened and when to the status reporting task.

A personal computer may also run several tasks that are actually separate programs. For example, a user might have the following programs running at the same time under a multitasking operating system such as OS/2 or Unix:

- A word processor printing a long document
- A spreadsheet for creating a table to be attached to the document
- A scratchpad accessory for jotting down notes from a telephone conversation
- A calculator for making a few quick computations on the data received by telephone
- A communications program for retrieving historical information from a remote financial or economic database

Here again, you want to suspend tasks when they are done or have gone as far as they can without more input or other external events. Suppose, for example, that you have reached the point in the spreadsheet where you need the historical data. Or perhaps the word processor has printed everything up to the table, or the communications program has encountered transmission problems. You will then want to resume the task later without any problems. You will also want to move information from one task to another. For example, you must move the calculator's results and the information retrieved from the remote database into the spreadsheet. You must then move the spreadsheet's output to the word processor.

Tasking thus has many advantages. It allows you to:

- Do several things simultaneously. You can switch from one task to another when necessary or when the original task no longer requires your attention.
- Change one task without changing the others. For example, in the energy management system, you could change inputs, outputs, alarm handling methods, or control techniques without affecting the other tasks. Similarly, you could add inputs or improved outputs. You could also add remote dialing, a tape backup, or communications capabilities.
- Pinpoint problems within a single task. You can use tasking to isolate problems to part of a large system. A simple approach is to keep replacing tasks with dummy versions until the error disappears.
- Incorporate tasks from the operating system or outside sources. Tasks such as a keyboard handler may be common to many applications.

Intel introduced tasking features in the 80286 and expanded them in the 80386. This is not to say that one could not do tasking on previous processors such as the 8086 and 8088. However, those processors did not provide explicit support such as special instructions and built-in data structures and registers. The built-in features execute faster and provide more standardization at the cost of some flexibility. Their use also ensures compatibility with software from Intel and many other sources.

Note, however, that the 80386's tasking features are not essential. One can do tasking without using them. In systems with simple needs, they may create unnecessary complexity and overhead. The mere fact that they exist does not make their use either essential or desirable. They provide a generalized framework that may be overkill for some applications.

80386 TASKING FEATURES

The 80386 provides the following task-related features:

- Task state segments.
- Task state segment descriptors.
- Task register.
- Task gate descriptors.
- The NT (nested task) flag in the extended flag register for use in returning from tasks that have been called by other tasks. NT is bit 14 of EFLAGS (see Figure 2-3).

80386 Task Management

31	23	15	7	0													
I/O MAP BASE		0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0												T	64		
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																LDT	60
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																GS	5C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																FS	58
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																DS	54
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																SS	50
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																CS	4C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																ES	48
																EDI	44
																ESI	40
																EBP	3C
																ESP	38
																EBX	34
																EDX	30
																ECX	2C
																EAX	28
																EFLAGS	24
																INSTRUCTION POINTER (EIP)	20
																CR3 (PDPR)	1C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																SS2	18
																ESP2	14
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																SS1	10
																ESP1	0C
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																SS0	8
																ESP0	4
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0																BACK LINK TO PREVIOUS TSS	0

NOTE: 0 MEANS INTEL RESERVED. DO NOT DEFINE.

Figure 6-1
32-bit task state segment.

Tasking applies in protected mode (including V86 mode) but not in real mode. V86 tasks must have the VM bit (in EFLAGS) set to 1 and must run at privilege level 3. A virtual machine monitor must handle special V86 exceptions such as INTs used to enter MS-DOS.

Task State Segments

Task state segments hold a task's current status in a predefined form. You may compare them to a business' balance sheet or a team's roster with up-to-date statistics. The existence of the TSS allows an operating system to readily change a task's state. The OS can activate it, suspend it, or terminate it (or kill it, if you have a violent nature) by using its task state segment. Task state segment descriptors define tasks. The task register points to the task state segment for the current task. Task gate descriptors provide indirect, protected access to task state segments. They do privilege checking much like the call gates described in Chapter 5.

Figure 6-1 shows the fields in a minimum task state segment. Note that even this segment is large and complex, occupying at least 104 bytes of memory (26 double words). It contains two types of information:

- Dynamic information that the processor updates each time it switches to another task. This includes the user registers or machine state — general-purpose registers, segment registers, flags, and instruction pointer. It also includes the selector for the task state segment of the previously executing task if a return to that task is expected.
- Static information (*software state*) that is a permanent part of the task's environment. This includes the selector for the task's local descriptor table, the base address of its page directory, pointers to the stacks for privilege levels 0 through 2, the debug trap bit, and the I/O map base. We will discuss the I/O map base (used to access the I/O permission bit map) later in this chapter.

The task state segment thus includes both a task's overall environment and its current status. Loading the task state segment not only replaces the usual initialization of registers that starts a task, but it also allows a suspended task to resume with its old status. This greatly simplifies the task scheduler's job. The disadvantage of the approach is that it introduces extra overhead for simple tasks without extensive status information.

A task state segment may contain more information than Figure 6-1 shows. Figure 6-2 is an extended task state segment with an optional software state and I/O permission bit map (used to give a task access to a limited set of I/O devices). We will discuss the I/O permission bit map (or I/O guard map) later in this chapter. The optional software state depends on the operating system, rather than being 80386-defined. It may, for example, contain the current state of a numeric coprocessor (see Chapter 8), permission bits defined by an operating system such as Unix, scheduling priority, ac-

80386 Task Management

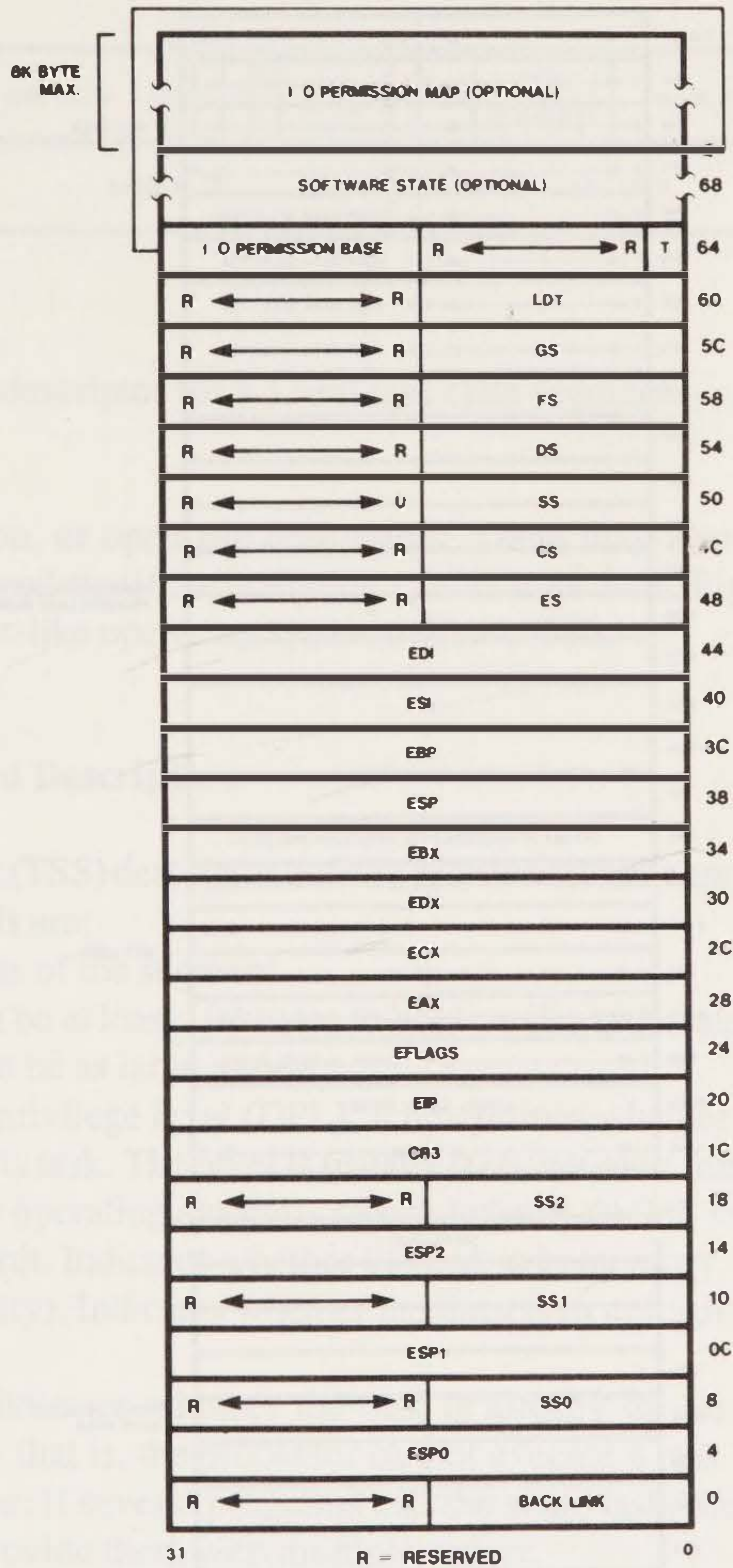


Figure 6-2
An extended task state segment.

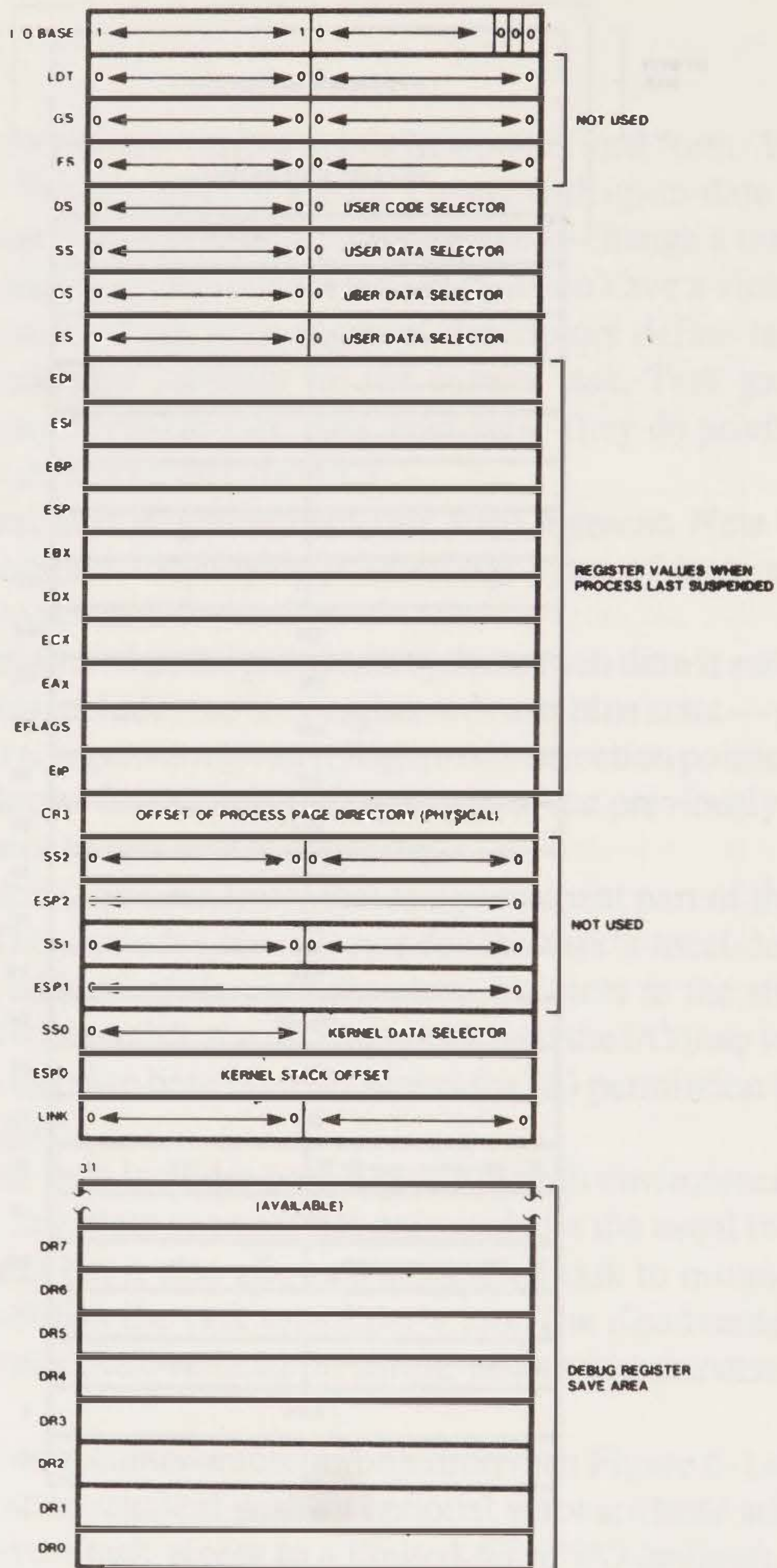


Figure 6-3

Process task state segment for a Unix-like operating system U/386.

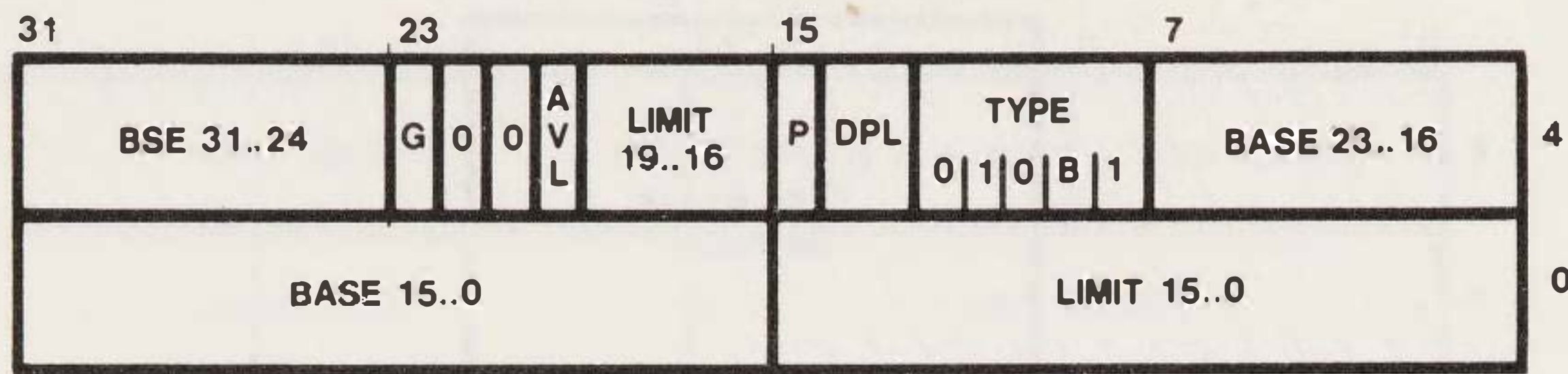


Figure 6-4

Task state segment descriptor for a 32-bit task state segment.

counting information, or open file descriptors. There may also be places for debug registers and other facilities if the operating system uses them. Figure 6-3 shows a TSS for an example Unix-like operating system called U/386.

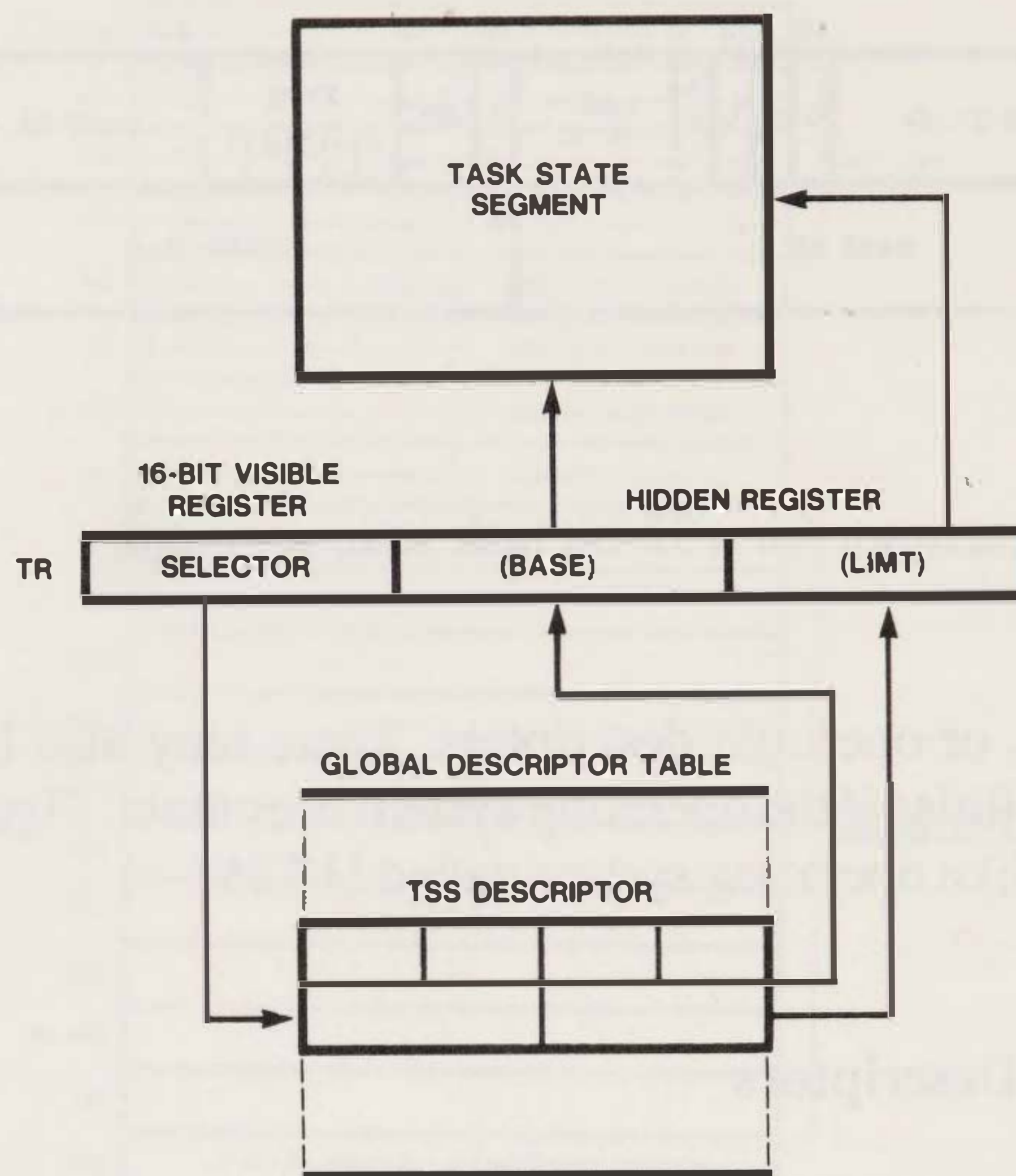
Task State Segment Descriptors

A task state segment (TSS) descriptor defines each task state segment. Figure 6-4 shows its format. The fields are:

- Base address of the segment.
- Limit (must be at least 103 bytes to hold a valid task state segment). The segment can be as large as 64K.
- Descriptor privilege level (DPL). It determines whether other tasks can switch to this task. The level is usually 0 so that only trusted procedures (such as the operating system's scheduler) can do task switching.
- P (present) bit. Indicates whether the task is in memory.
- G (granularity). Indicates whether the limit is in units of bytes (0) or 4K bytes (1).
- B (busy). Indicates whether the task is already in use. Tasks are not reentrant — that is, the processor cannot execute a task while the same task is active. If several programs call the same task, the operating system must provide them with multiple copies.

TSS descriptors have the following special features:

- They do not allow for reading or writing of the TSS. The only way to do either is to create another descriptor that redefines the TSS as a data segment. Simply loading the TSS descriptor into a segment register causes an exception.

**Figure 6-5**

Task register and its relationships to the task state segment and task state segment descriptor.

- They must reside in the global descriptor table. TSS's cannot be in a local descriptor table.

Task Register

The task register contains the selector for the currently executing task. Like a segment register, this register has visible and hidden parts. The hidden parts contain the task state segment's base and limit, derived from the descriptor when its selector is loaded. Figure 6-5 shows the relationships among the elements that define a task.

There are special instructions for loading and storing the task register. They are:

- LTR loads the task register from a 16-bit general-purpose register or from a memory word.

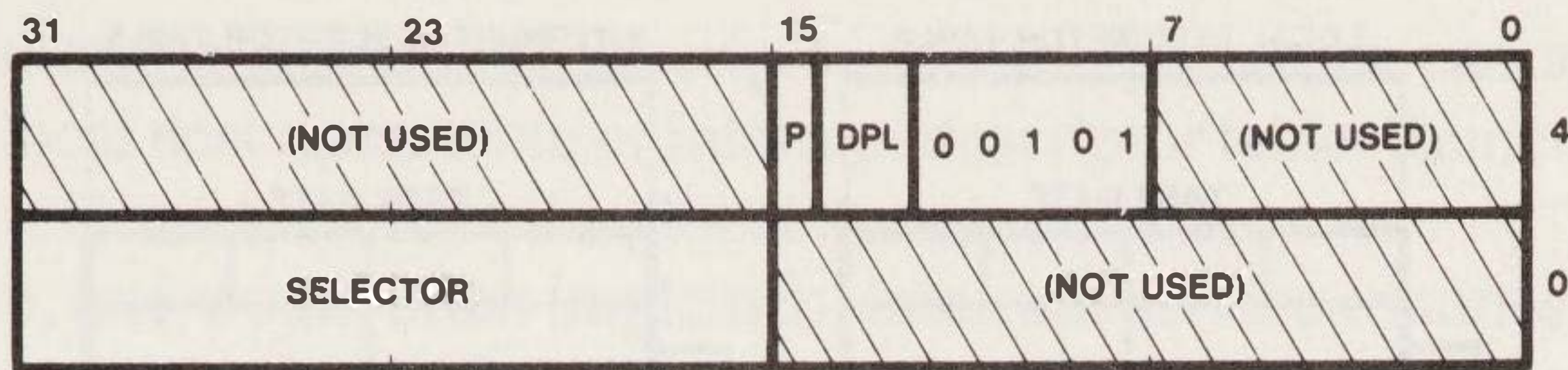


Figure 6-6
Task gate descriptor.

- STR stores the task register in a 16-bit general-purpose register or in a memory word.

Only procedures running at privilege level 0 (usually operating system procedures) in protected mode can execute LTR. STR is not privileged. These are always 16-bit operations. The operand-size attribute has no effect on them. Remember that in practice, the processor transfers the hidden part of the task register or descriptor along with the visible part.

The operating system generally uses LTR only to initialize the task register. Task switches then change the value as needed.

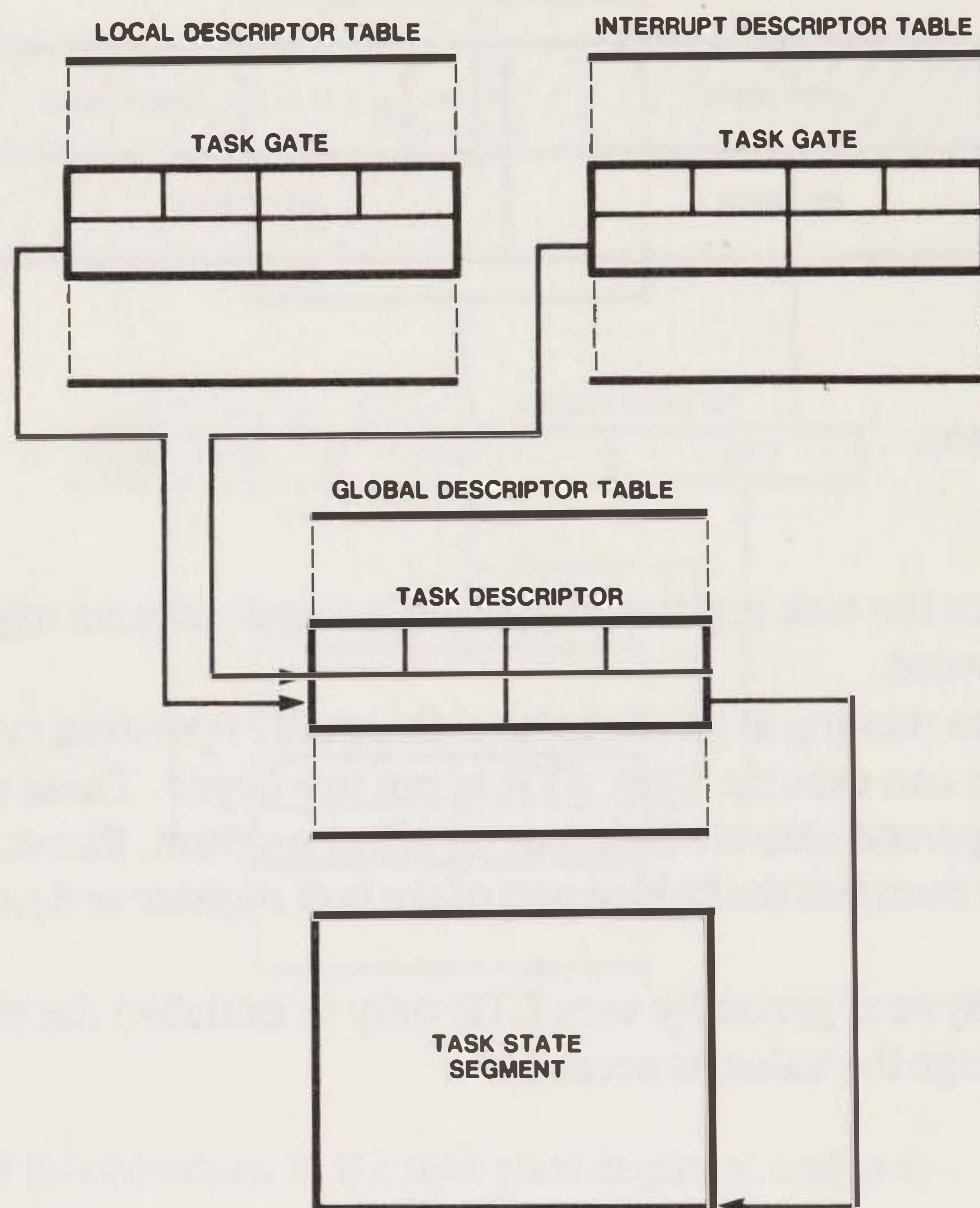
Task Gate Descriptors

A task gate descriptor provides an indirect, guarded entry point to a task state segment. Figure 6-6 shows a task gate's format. It contains the following fields:

- Selector for a task state segment descriptor.
- P (present) bit that indicates whether the descriptor is currently in memory.
- DPL (descriptor privilege level). This is the privilege level required to use the gate.

Note the differences between a task gate and a call gate:

- A task gate refers to a task state segment descriptor, not to an actual entry point. Thus a task gate has no offset field. There is an extra level of indirection here.
- Task gates do not allow parameter passing, so they do not contain a parameter count field. Since the stack pointer changes during a task switch, tasks cannot communicate through the stack. They must com-

**Figure 6-7**

Indirect access to a task descriptor through several task gates.

municate instead through shared memory areas. The precise method, of course, is OS-dependent.

Task gates provide flexibility to systems for the following reasons:

- Several task gates may select the same descriptor, thus providing entry points at different privilege levels or through the local descriptor table or interrupt descriptor table.
- Task switches can be limited to specific tasks through entries in the local descriptor table.
- Tasks can be activated by interrupts or exceptions through entries in the interrupt descriptor table.

Figure 6-7 shows a single task descriptor that the processor can access directly or through the local descriptor table or interrupt descriptor table. Remember that the task descriptor itself must reside in the global descriptor table.

In practice, one would often implement interrupt service routines as tasks because they require their own contexts. Since an interrupt can occur at any time, its handler must operate independently of the suspended program. An interrupt handler usually goes through what amounts to a task switch anyway. That is, it saves all the registers and then does some initialization. Exception handlers, on the other hand, often must use the context of the task that incurred them to correct or describe conditions. Note that interrupt and exception handlers implemented as tasks need not save and restore registers, as the task switch does that automatically.

TASK SWITCHING

A task switch can occur through a task state segment descriptor or a task gate. As noted above, the processor can access a task gate from the current task or via an interrupt or exception. A task switch also occurs when the current task does a return with its NT (nested task) flag set.

How does the processor know whether to do a task switch? It knows either from the type of descriptor referenced or from the NT flag. Remember that each descriptor has a type field. The reference can be either to a task state segment descriptor or to a task gate.

A task switch proceeds as follows, assuming that the new task is present and has a valid limit:

- The processor checks whether privilege levels allow the switch. If they do not, a general privilege exception occurs (see Chapter 7).
- The processor saves the current task's state. The base address of the current task state segment is in the hidden part of the task register. The processor saves all general-purpose registers and segment registers, the flag register, and the instruction pointer in the task state segment. This is roughly equivalent to a series of MOVs plus an interrupt response.
- The processor loads the task register with the selector of the new task's TSS descriptor. It also loads the hidden part of the descriptor at this time.
- The processor loads the new task's state from its TSS and executes it. The state includes all general-purpose and segment registers, the local descriptor table register, the flags, and the page directory base register.

Loading it is roughly equivalent to a series of MOVs plus an interrupt return.

The instructions that can cause a task switch are JMP, CALL, and IRET. Interrupts and exceptions can also cause a task switch (through the interrupt descriptor table as shown in Figure 6-7).

A task switch resembles an interrupt response. The processor always saves the previous task's state. If that task is resumed, it starts after the instruction that caused the switch. The registers have the values they had before the switch, much as they would if an interrupt had occurred.

A task switch sets the TS (task switched, probably not what you were thinking) bit in control register 0 (bit 3; see Figure 2-4). Its main use is to show whether the numeric coprocessor's state is related to the current task. If TS is set, a numeric coprocessor instruction causes an exception. The handler must then determine whether a different task is now active from the one that did the last instruction. If so, the handler must save the coprocessor's state in the previous task's TSS. This allows the operating system to avoid saving the coprocessor's state if intervening tasks do not use it anyway. Chapter 8 describes coprocessors in more detail. The tradeoff here is a good one, since relatively few tasks use a numeric coprocessor in most systems.

Note that the new task's privilege level is not related to the old task's. After all, tasks have their own address spaces and task state segments. Furthermore, privilege rules prevent improper access to a task state segment. The new task's execution level is the RPL of the code segment selector value loaded from the task state segment.

Task switches do not change systemwide resources. These are control register 0, the global and interrupt descriptor table registers, and control register 2 (the page fault linear address).

Task switches have both advantages and disadvantages. On the positive side, they provide a standard way to switch control between independent programs. The new task has its own context. It does not share limited resources such as stack space with the old task. On the negative side, task switches may be time-consuming and inefficient. For example, a simple keyboard interrupt handler would only need to save and restore a few registers rather than the complete machine state used in a task switch. The same would also apply to serial communications, printer, or real-time clock interrupts. Such handlers would therefore probably not be implemented as tasks.

TASK LINKING

How does a task return control to its predecessor? Note that the entire address space may have changed, so a link in the stack is not sufficient. Instead, the 80386 fills the back link of the new task's TSS (see Figure 6-1) with the selector of the old task's TSS. It also sets the NT bit in the new task's flag register, indicating that the back-link field is valid. The new task must end with IRET. The processor will then switch back to the task in the back-link field if the NT flag is set. This applies to new tasks entered through CALLs or through interrupts or exceptions but not to those entered through JMPs.

One problem is that a chain of back links may become circular. What happens if the processor eventually tries to return to the original task? The chain may, of course, be quite long so its circularity is far from obvious. For example, such a chain could result from a series of interrupts in a multilevel nested system.

The solution is to use the B (busy) bit of the task state segment descriptor (see Figure 6-4). Note that this bit is part of the type information — a busy task is actually a different type from a nonbusy task. The procedure is as follows:

1. The processor automatically sets the busy bit of each new task when it is activated.
2. When switching from a task, the processor clears its busy bit if it is not to be placed on the back-link chain. This is the case if the switch occurs because of a JMP or IRET instruction. Otherwise, the B bit remains set. This happens if the switch is the result of a CALL, an interrupt, or an exception. B thus indicates that the task is either currently active or will be resumed later.
3. When switching to a task, the processor causes an exception if the busy bit is set. Thus a task cannot switch to any task on the back-link chain, no matter how long the chain is.

What happens if the system must remove a task from the back-link chain? Suppose, for example, that an error occurs and the task cannot be resumed. Now the operating system must change the back-link field in the TSS of the next task on the chain. It must then clear the B bit in the TSS descriptor of the task that must be removed. If the error is a fatal one such as a double bus error or a memory parity error, the system must remove all links before exiting. Otherwise, the system could leave tasks in the busy state erroneously, and other tasks could not transfer control to them. This is one aspect of the general problem of removing “dead” tasks from multitasking systems.

Note also that the operating system should clear unused back links. The clearing will make accidental references to them cause exceptions rather than task switches with unpredictable results.

TASK ADDRESS SPACES

Several parameters define a task's address space. They are:

- The global descriptor table. All tasks can access addresses defined through this table.
- The local descriptor table. Only a specific task can access addresses defined through this table unless several tasks share a descriptor table or descriptors actually refer to the same address space (that is, they are *aliases* of each other).

Since the TSS (Figure 6-1) includes values for both the local descriptor table register and the page directory base register, tasks may have separate or shared address spaces at any level. Modules may cooperate through shared address spaces. The operating system may use the same page directory for all tasks or may assign them different directories.

Tasks can share address spaces in the following ways:

- By using entries in the global descriptor table. All tasks have access to these descriptors. No exclusion or restriction is possible.
- By sharing a local descriptor table. This approach is more restrictive than the one based on the global descriptor table. Some tasks may have access to the space whereas others may not. However, all local descriptors are always part of the shared area.
- By using entries from different local descriptor tables that point to the same linear address space. As just noted, we commonly call such descriptors *aliases* since they are different names for the same thing. This method of sharing addresses is even more restrictive than the sharing of local descriptor tables, as it may involve only a few descriptors. Other entries in the local descriptor tables may point to distinct linear addresses.

Programs can share data through a shared memory facility. The facility could consist of a public region implemented using shared page tables.

I/O PRIVILEGE LEVELS

One problem with multitasking and multiuser systems is managing the use of shared I/O devices. For example, suppose we have a multitasking personal computer with a typical complement of I/O devices: keyboard, video display, disk, and printer. The operating system must control access to these devices. Otherwise, several tasks could try to use them at the same time. You could, for example, find compiler warning or error messages in the middle of a typed document. Or the results of a spreadsheet could appear in a program listing or on mailing labels. Similarly, consecutive keystrokes could end up serving as inputs to different programs.

The way to avoid such confusion is to assign each task an input/output privilege level (IOPL) higher than its operating (current) privilege level. Then an attempt by the task to do I/O causes a general protection exception. The operating system takes control and manages the I/O. IOPL is bits 12 and 13 of the extended flags (see Figure 2-3). Hence its value is part of the status in the TSS (Figure 6-1).

For example, in a simple user/supervisor system with privilege levels 0 and 3, all user tasks would have IOPL = 0. Thus an attempt by a user task to do I/O would cause a supervisor call. Of course, the supervisor action would be transparent as far as user tasks were concerned. The only noticeable effect would be long, irregular execution times for trapped instructions (IN, INS, OUT, OUTS, INT n, IRET, PUSHF, POPF, STI, and CLI). IRET, PUSHF, and POPF are included here because they can change IF (and hence IOPL).

IOPL and I/O permission maps (our next subject) apply also in V86 mode but not in real mode. In V86 mode, using IOPL can keep 8086 applications programs from disabling interrupts for long periods, reprogramming disk controllers, or accessing other hardware devices directly. The V86 monitor can then emulate the functions and devices without interfering with other tasks. Emulation can also allow programs that manipulate hardware directly to run on newer computers. New machines may, for example, have more powerful interrupt, CRT, DMA, and disk controllers than older computers. The monitor can direct manipulations to virtual I/O devices, which then convert them into operations for the new devices.

I/O PERMISSION MAPS

How can we allow tasks to use some I/O devices but not others? For example, we might want to allow:

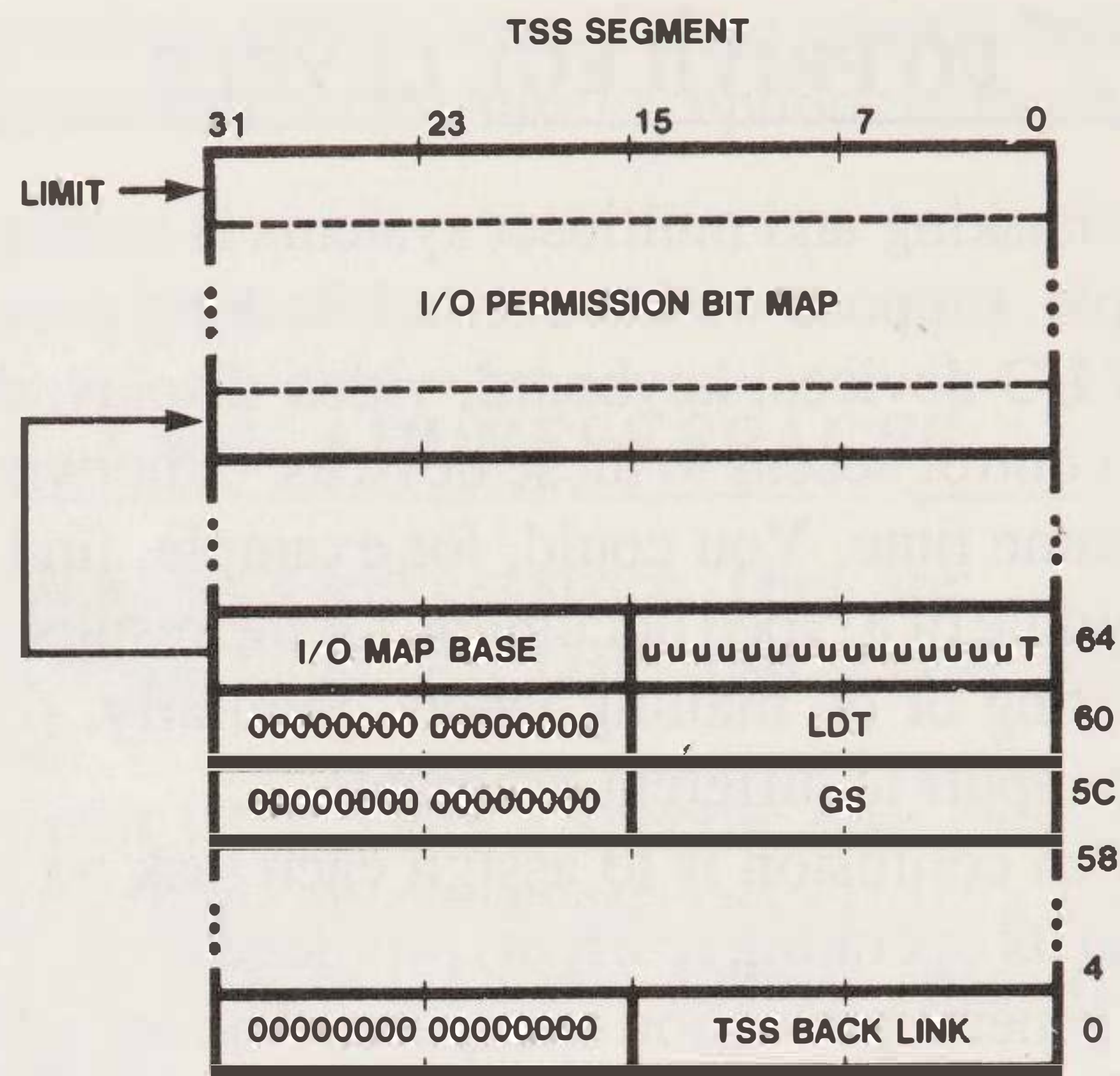


Figure 6-8
I/O address permission bit map.

- Each user on a multiuser system to access his or her own console and printer but not anyone else's or systemwide devices.
- A particular task to access a specific I/O device such as a high-speed analog channel or a signal or image processing board. Obviously, this is particularly important for real-time devices where the overhead caused by an exception and an OS call would be intolerable.
- Foreground (high-priority interactive) tasks to access the keyboard, display, or printer on a single-user machine, while background (low-priority batch) tasks cannot. The operating system may provide *virtual devices* (usually disk files) for background tasks to use.
- V86 tasks to access certain I/O devices (such as a CRT controller or I/O port) directly but not others (such as disk, interrupt, or DMA controllers).

The solution is to use an I/O permission bit map (also called an *I/O guard map*). It is accessible from the task state segment as shown in Figure 6-8. The two highest-addressed bytes in the TSS contain a pointer to the I/O permission map. Each bit in the map corresponds to an I/O port byte address. If the bit's value is 1, the task cannot access the port. That is, IOPL applies. If the bit is 0, the port is accessible despite IOPL's value.

For example, suppose a task tries to do output to port 35 (decimal). If the task's IOPL is more privileged than its CPL, the processor examines the I/O permission bit map to see if I/O may be allowed anyway. The controlling bit is bit 3 in the byte 4 beyond the map's base address. That is, bit 35 is equivalent to bit 3 of byte 4. I/O is allowed if that bit is 0.

Note the following about I/O permission bit maps:

- Their sizes must be included in the limit specified in the TSS descriptor.
- They extend only as far as the TSS descriptor limit. Any ports that are not in the map are assumed to be inaccessible. That is, their permission bits are 1 by default. So you need not include a map if IOPL is to apply to a task. Nor do you have to specify ports beyond the highest numbered accessible one.
- Operations involving multibyte ports are allowed only if all bytes have 0 permission bits.
- They must end with an all-1s pad byte. It ensures that the processor can read the last byte. Reading occurs on a word basis.
- If a task state segment has no bit map, its map base pointer should be set to the segment limit. It should not be zero! A sure way to define a null permission map is to set the base pointer to FFFF hex. Figure 6-9 shows both explicit and null maps.

I/O permission bit maps are a new feature of the 80386. Previous processors did not have them.

INITIALIZATION OF TASKING SYSTEMS

You can use Intel's BLD386 or other similar tools to create task state segments, task state segment descriptors, task gates, and entries in global, local, and interrupt descriptor tables. BLD386 can obtain initialization information from user input or from a specified input module. It will even create TSSs automatically when the user does not provide specific task definitions. Of course, you can also create all the structures manually if you have the will power.

The initialization information for TSSs can include:

- Task name
- TSS descriptor
- Task LDT selector
- Initial code and data segment values

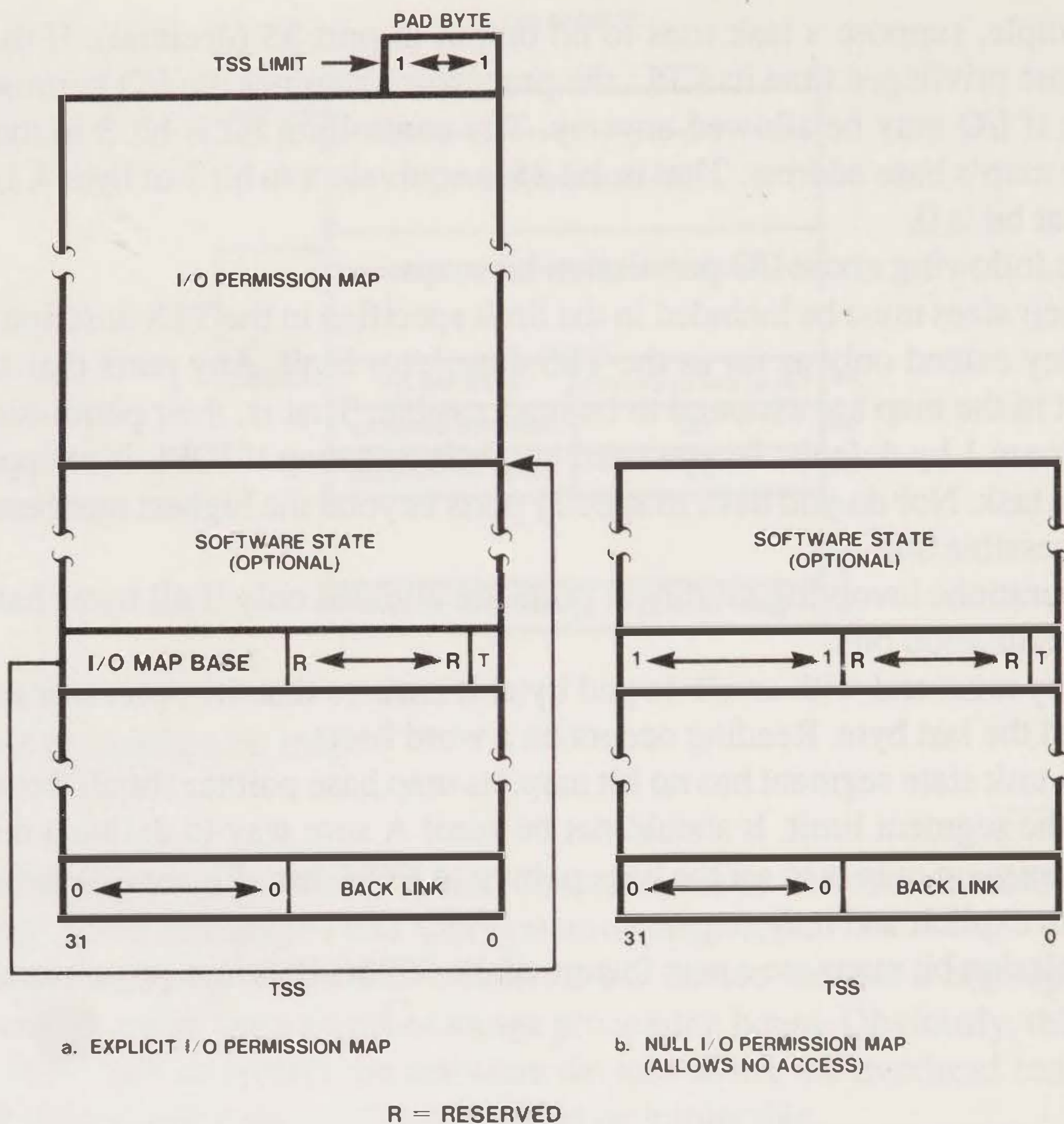


Figure 6-9

Explicit and null I/O permission maps.

- Stack selectors
- I/O privilege, interrupt status, and trap flag values
- Base address
- Size

BLD386 creates a TSS descriptor for each TSS. You can specify the values for:

- Present bit
- Descriptor privilege level
- Base address


```
MULTITASK;

  SEGMENT
    NUCLEUS (DPL = 0) ;

  TABLE
    COMMONLDT (ENTRY = (MOD1, MOD2, SDATA)) ;

  TASK
    TASK1BLOCK (OBJECT = MOD1,                --DPL=3
                LDT = COMMONLDT,
                STACKS = (NUCLEUS.STACK1)) ,    --DPL=0

    TASK2BLOCK (OBJECT = MOD2,                --DPL=3
                LDT = COMMONLDT,
                STACKS = (NUCLEUS.STACK2)) ;    --DPL=0

  GATE
    TASK1GATE (TASK, ENTRY = TASK1BLOCK) ;
    TASK2GATE (TASK, ENTRY = TASK2BLOCK) ;

  TABLE
    GDT (ENTRY = (NUCLEUS,
                  COMMONLDT,
                  TASK1BLOCK,
                  TASK1GATE,
                  TASK2BLOCK,
                  TASK2GATE)) ;

END
```

Figure 6-10

Example BLD386 program for a two-task module.

- Limit

BLD386 can also create task gates. You can specify:

- Gate name
- Entry point
- Descriptor privilege level
- Present bit
- Type (80286 or 80386)

Figure 6-10 shows an example BLD386 program for a two-task module. The tasks share a local descriptor table COMMONLDT and a global data area SDATA (for shared data). The LDT contains descriptors for all segments in modules MOD1 and MOD2. Both tasks have stack segments defined in module NUCLEUS. The global descriptor table, defined at the bottom, has entries corresponding to all segments in the module NUCLEUS, the local descriptor table, the two task state segments, and the two task gates. Note that we need only specify a few parameters. BLD386 takes care of the rest by default.

You must also create a dummy TSS and a valid TSS descriptor for the operating system to use in the first task switch. The dummy serves as the nonexistent previous

task in this case. The operating system must use LTR to load the dummy's selector into the task register. It can then start the first task with a JMP TSS instruction. The 80386 will write the machine state into the dummy TSS just as though it had switched from a real task. Note that you must use JMP TSS to avoid creating a back link.

A V86 task must have the following special initialization:

- VM bit in EFLAGS set to 1
- CS selector field set to the linear base address of the task's initial code segment divided by 16
- IP field set to the task's entry point
- IOPL field in EFLAGS set to 3 if the task can access the I flag and to 0 otherwise
- LDT selector field set to 0 unless an interrupt or exception procedure uses an LDT.

SUMMARY

The 80386 has many features aimed at efficient implementation of independent programs or tasks. Each task's current state is in a data structure called a task state segment (TSS). The TSS contains initial values for the general-purpose registers, segment registers, local descriptor table register, flags, instruction pointer, page directory, base register, and stack pointers (and stack segment registers) for all privilege levels at which the task may run. The TSS also contains a back link to the previous TSS. The link is valid if another task called the current task and must regain control from it on termination.

Each TSS has a corresponding descriptor in the global descriptor table. The descriptor contains a base address, a limit, a privilege level, and a busy bit. The task register contains the selector for the currently active TSS. The register's hidden part contains the descriptor's base address and limit.

Task gates provide indirect, guarded entry points to task state segments. A task gate contains a selector for a TSS descriptor, a present bit, and a privilege level. The use of task gates allows interrupt or exception handlers to be implemented as tasks. This approach is particularly applicable to complex interrupt handlers; it is less well-suited to simple interrupt handlers and most exception handlers.

A task switch involves saving the current task's status in its TSS and loading the new task's status from its TSS. Such a switch occurs when a task jumps to or calls a TSS descriptor or a task gate. It also occurs when a task does a return with its NT (nested

task) flag set. Task switches work much like a transfer of control to an interrupt service routine. Tasks can be linked to any depth through the back-link field in their TSSs.

Tasks can have distinct (private) or shared address spaces. The usual way to obtain private address spaces is through separate local descriptor tables. The usual ways to share address spaces are through the global descriptor table, common local descriptor tables, or descriptors that point to the same linear address space (aliases). These approaches differ in terms of how much you can limit the sharing.

Tasks may have restricted ability to do I/O. The I/O privilege level (IOPL) in the flag register determines the minimum privilege level at which I/O is permitted. Tasks running at less privileged levels can do I/O only under the control of more privileged levels. However, the I/O permission bit map in the task state segment can remove this restriction for specific ports. Restrictions on I/O are essential for managing shared devices on multitasking and multiuser systems. They also allow newer systems to run programs that assume older I/O configurations.

80386 Exceptions and Debugging Features

No rule is so general, which admits not some exception.
Burton, *The Anatomy of Melancholy*

I never make exceptions. An exception disproves the rule.
Sir Arthur Conan Doyle, *The Sign of Four*

*The first 90 percent of the code accounts for the first
90 percent of the development time. The remaining
10 percent of the code accounts for the other 90 percent of
the development time.*
Tom Cargill

This chapter covers 80386 exceptions and debugging features. Exceptions, as noted earlier in a discussion of interrupts, are internal conditions or instructions that make

the 80386 suspend its normal activities and do special routines. Typical causes are overflow, division by zero, page faults, and protection violations. The chapter describes the sources of exceptions, their classification and identification numbers, interrupt descriptor tables, interrupt and trap gates, interrupt tasks and procedures, error codes, and exception conditions. The last section presents the 80386's special debugging features.

NEW 80386 FEATURES

Exceptions work much the same on the 80386 as on its predecessor, the 80286. The differences are:

- The use of interrupt 1 for general debugging exceptions rather than just for single-step inputs. This change reflects the introduction of hardware debugging features in the 80386.
- The introduction of exception types (faults, traps, and aborts).
- The addition of page fault and coprocessor error exceptions. Paging is a new feature in the 80386, and page fault exceptions are the key to creating demand-paged virtual memory systems.
- Revised definition of double faults so that they occur only if the processor detects a serious exception while processing another serious exception. Double faults are thus far less likely on the 80386 than on the 80286.

The 80386 also has new or revised exception conditions resulting from its larger task state segments and the addition of two new segment registers (FS and GS).

The 80386 has greatly expanded hardware debugging features. The additions are four debug address registers, along with control and status registers. They allow debuggers to specify instruction or data breakpoints in hardware rather than in software.

SOURCES OF EXCEPTIONS

The 80386 handles exceptions much as it handles the external interrupts discussed in Chapter 4. In fact, we may consider interrupts as special cases of the more general category of exceptions. Table 7-1 summarizes 80386 exceptions. This table overlaps considerably with the interrupt summary in Table 4-3.

There are two types of exceptions:

- Processor detected.

Table 7-1
Summary of 80386 Exceptions

Description	Interrupt Number	Return Address Points to Faulting Instruction	Exception Type	Function That Can Generate the Exception
Divide error	0	YES	FAULT	DIV, IDIV
Debug exceptions	1	*1	*1	Any instruction
Breakpoint	3	NO	TRAP	One-byte INT 3
Overflow	4	NO	TRAP	INTO
Bounds check	5	YES	FAULT	BOUND
Invalid opcode	6	YES	FAULT	Any illegal instruction
Coprocessor not available	7	YES	FAULT	ESC, WAIT
Double fault	8	YES	ABORT	Any instruction that can generate an exception
Coprocessor Segment Overrun	9	NO	ABORT	Any operand of an ESC instruction that wraps around the end of a segment.
Invalid TSS	10	YES	FAULT ²	JMP, CALL, IRET, any interrupt
Segment not present	11	YES	FAULT	Any segment-register modifier
Stack exception	12	YES	FAULT	Any memory reference thru SS
General Protection	13	YES	FAULT/ABORT ³	Any memory reference or code fetch
Page fault	14	YES	FAULT	Any memory reference or code fetch
Coprocessor error	16	YES	FAULT ⁴	ESC, WAIT
Two-byte SW Interrupt	0-255	NO	TRAP	INT n

- Programmed. These are the so-called *software interrupts* INTO, INT 3 (a special 1-byte instruction), INT n, and BOUND. They are instructions whose sole purpose is to cause an exception. Their most common uses are in data and address validation and in debugging (see the last section of this chapter).

We may further classify processor-detected exceptions as (in order of increasing seriousness) *faults*, *traps*, and *aborts*. Table 7-1 lists the type of each exception and in-

Table 7-2
Priority Among Simultaneous Exceptions and Interrupts

Priority	Class of Interrupt or Exception
HIGHEST	Faults except debug faults Trap instructions INTO, INT n, INT 3 Debug traps for this instruction Debug faults for next instruction
LOWEST	NMI interrupt INTR interrupt

icates whether the return address points to the faulting instruction. It also notes what instructions or references can cause the exception.

Most exceptions are faults (the least serious category). Faults (for example, breakpoints, divide errors, and not-present conditions) are correctable. The processor reports them before it finishes executing the instruction. It saves the instruction's address (code segment register and instruction pointer values) to allow restart. Note that unlike the Motorola 68010 and 68020, the 80386 cannot continue the instruction from the point at which it was interrupted. It can only start it over from the beginning.

Traps (for example, single-step and task-switch breakpoint) do not need to be restarted. In fact, restarting them would cause an endless loop. The processor reports them when it reaches the next instruction. It saves that instruction's address to allow program resumption. The next instruction may be the target of a jump.

Aborts (double faults and other serious hardware errors) do not allow for restart or resumption. The processor does not save an instruction address. The operating system must take complete control of the situation in this case.

Table 7-2 shows the priority among simultaneous interrupts and exceptions. The processor holds lower priority interrupts pending while it services the event with the highest priority. It discards lower priority exceptions but will rediscover them later upon eventual return to the point of interruption.

INTERRUPT DESCRIPTOR TABLE

We have already mentioned the interrupt descriptor table in Chapter 4. However, our discussion there dealt only with the 8086-like case in which the table consists of 4-byte

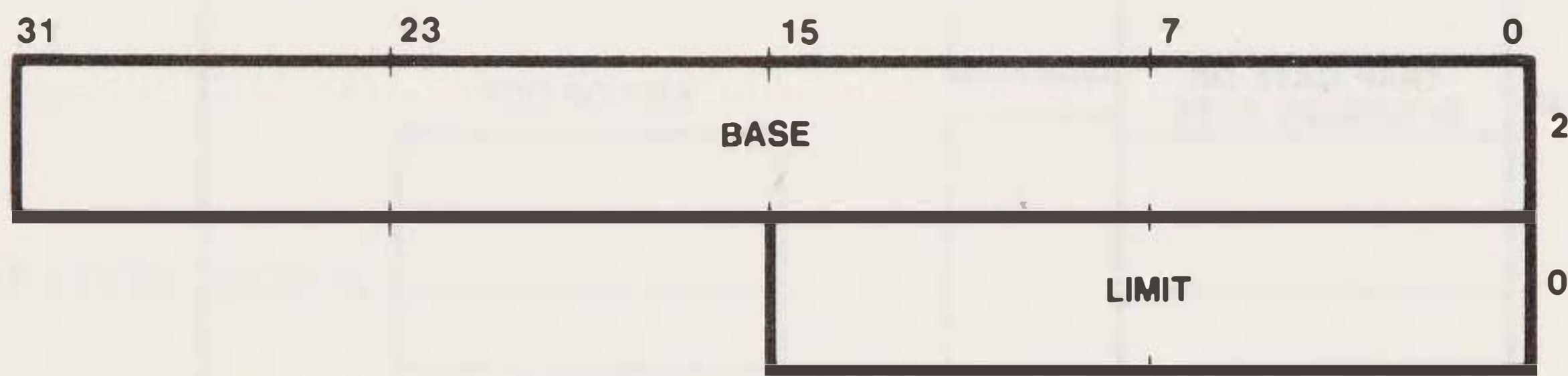


Figure 7-1

Interrupt descriptor table and interrupt descriptor table register.

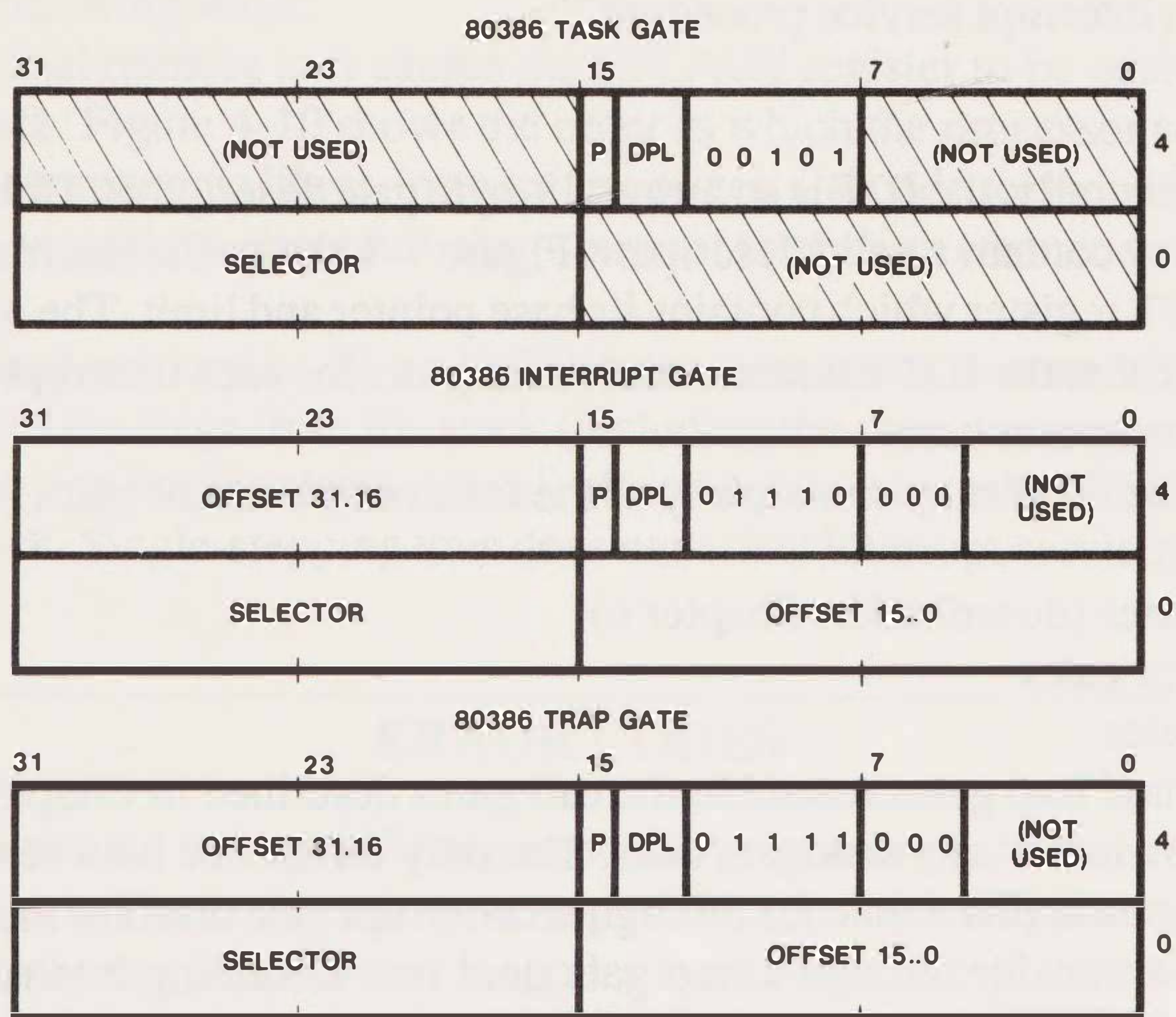


Figure 7-2

Descriptors for 80386 task, interrupt, and trap gates.

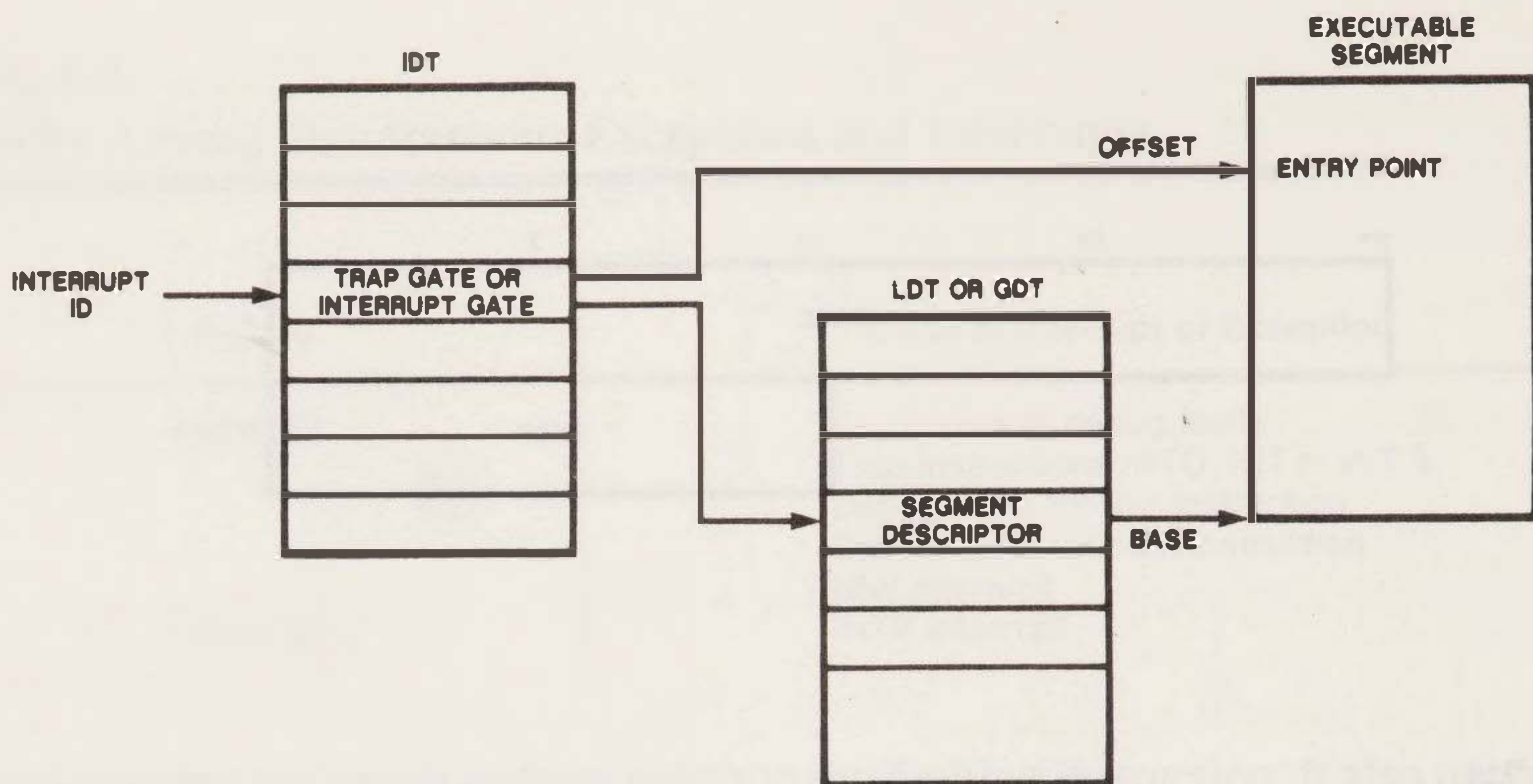


Figure 7-3

Vectoring to an interrupt service procedure.

entries. More generally, the IDT is an array of 8-byte gate descriptors. Unlike the GDT, its first entry may contain a valid descriptor. Figure 7-1 shows the interrupt descriptor table and the IDT register which contains its base pointer and limit. The operating system must ensure that the IDT contains valid descriptors for each interrupt or exception number that a computer uses.

The generalized IDT may contain any of the following kinds of gates (see Figure 7-2):

- Task gates (described in Chapter 6)
- Interrupt gates
- Trap gates

Interrupt gates and trap gates resemble the call gates described in Chapter 5. They do not cause task switches as a task gate does. The only difference between an interrupt gate and a trap gate is that a transfer through an interrupt gate disables maskable interrupts, whereas a transfer through a trap gate does not. Disabling interrupts prevents later interrupts from interfering with the current handler. As you can see from Figure 7-2, only a single bit in the status field differentiates between trap and interrupt gates.

Exception handlers are usually invoked through trap gates. Only ones that require interrupts to be disabled are invoked through interrupt gates.

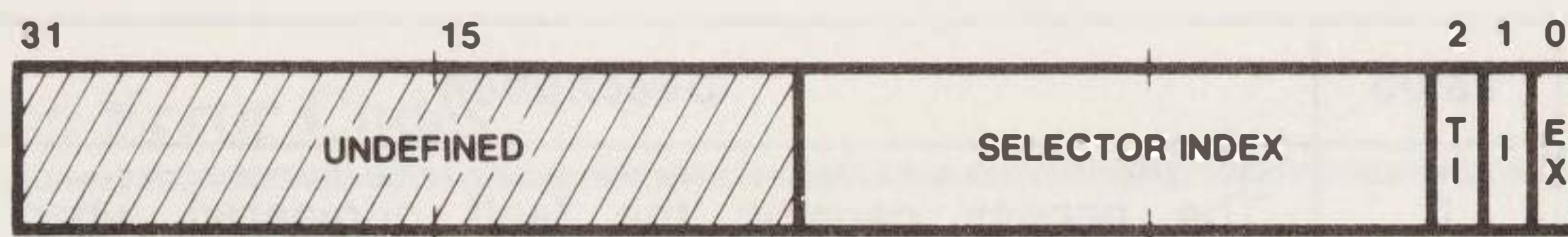


Figure 7-4

Exception error code format.

All three types of gates contain privilege levels. As mentioned in Chapter 6, exception handlers generally run at privilege level 0 in order to access operating system data.

Interrupt gates or trap gates cause an indirect jump to a procedure that executes in the current task's context as shown in Figure 7-3. Their descriptors may be in either the local or the global descriptor table, although exception handlers are usually in the GDT to avoid task dependence. The invoked procedure differs from an ordinary procedure in the following ways:

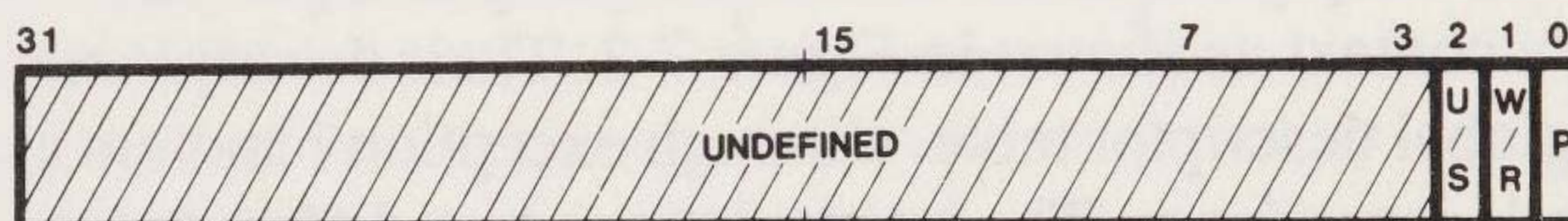
- The initial transfer to it causes the EFLAGS register to be pushed onto the stack. Figure 4-10 shows the order in which the processor saves the status components (flags, instruction pointer, and code segment register).
- Many exceptions also cause the processor to push an error code onto the stack.
- The usual exit is via a 32-bit IRET instruction rather than an RET. IRET retrieves the flags from the stack (including the previous state of IF).
- TF (the trap flag) is reset (cleared) after the processor saves EFLAGS on the stack. Single-stepping thus does not affect interrupt servicing.

ERROR CODES

For exceptions related to a specific segment, the processor pushes an error code onto the handler's stack. The code ends up on top of the status (see Figure 4-10). Figure 7-4 shows the usual error code format. It applies to all codes except those resulting from page faults. The fields are:

- Selector index — the upper 14 bits of the segment selector
- I (IDT) bit — 1 if the selector is a gate descriptor in the IDT, 0 otherwise.

Field	Value	Description
U/S	0	The access causing the fault originated when the processor was executing in supervisor mode.
	1	The access causing the fault originated when the processor was executing in user mode.
W/R	0	The access causing the fault was a read.
	1	The access causing the fault was a write.
P	0	The fault was caused by a not-present page.
	1	The fault was caused by a page-level protection violation.

**Figure 7-5**

Error code format for page fault errors.

- TI bit — if I is 0, 0 indicates that the error code refers to the global descriptor table and 1 indicates the local descriptor table. It is meaningless if I is 1.
- EXT bit — set (1) if an external event caused the exception, cleared (0) otherwise.

The error code for page faults is special (see Figure 7-5). It has 3 bits that indicate:

- Whether the processor was in supervisor or user mode (U/S field). 0 means supervisor, 1 user.
- Whether the problem occurred during a read or a write (W/R field). 0 means read, 1 write.
- Whether the problem was a not-present page or a protection violation (P field). 0 means a not-present page, 1 a page-level protection violation.

Table 7-3 summarizes the 80386's exception error codes. Handlers must pop error codes from the stack before resuming the suspended program.

Table 7-3
80386 Exception Error Codes

Description	Interrupt Number	Error Code
Divide error	0	No
Debug exceptions	1	No
Breakpoint	3	No
Overflow	4	No
Bounds check	5	No
Invalid opcode	6	No
Coprocessor not available	7	No
System error	8	Yes (always 0)
Coprocessor Segment Overrun	9	No
Invalid TSS	10	Yes
Segment not present	11	Yes
Stack exception	12	Yes
General protection fault	13	Yes
Page fault	14	Yes
Coprocessor error	16	No
Two-byte SW interrupt	0-255	No

EXCEPTION CONDITIONS

The general exception conditions are:

- 0 — divide error (divisor is zero during DIV or IDIV)
- 1 — debug exceptions — breakpoints, single-step, and other debugging conditions
- 3 — 1-byte breakpoint (INT 3)
- 4 — overflow (INTO)
- 5 — bounds check (BOUND)
- 6 — invalid operation code
- 7 — coprocessor not available
- 8 — double fault
- 9 — coprocessor segment overrun
- 10 — invalid task state segment
- 11 — segment not present
- 12 — stack exception
- 13 — general protection exception

14 — page fault

16 — coprocessor error

Note: Remember that the nonmaskable interrupt input uses identification code 2 and that Intel has reserved interrupts 15 and 17 through 31 without assigning any current functions to them (see Table 4-3).

The following exceptions are useful for applications programmers:

1. Interrupt 0 — divide error. This exception occurs during a divide instruction (DIV or IDIV) if the divisor is 0.
2. Interrupt 3 — breakpoint. This is the standard software interrupt. Its 1-byte length makes it handy in debugging. By replacing the first byte of an instruction with it, you can make the program stop and return control to a monitor or operating system. Debuggers often use INT 3 to set breakpoints. Note that this approach does not work if the program is in ROM.
3. Interrupt 4 — overflow. This instruction is used to recognize arithmetic overflow. It causes an exception if the Overflow flag is set (1).
4. Interrupt 5 — bounds check. This fault occurs if the processor finds an array reference outside the specified limits. The BOUND instruction activates it.

Interrupt 1 depends on the debug registers discussed later in this chapter. Interrupts 7, 9, and 16 depend on the coprocessor interface and hence are covered in Chapter 8.

Invalid Operation Code Exception (Interrupt 6)

This fault occurs when the execution unit detects an invalid operation code. It does not occur at instruction prefetch. No error code is pushed onto the stack. The fault also occurs when the execution unit detects an illegal type of operand such as a register as the target of a jump.

DOUBLE FAULTS

A double fault is caused by a serious exception occurring while the processor is handling a previous serious exception. To determine when a double fault occurs, we must first divide exceptions into three classes as shown in Table 7-4, namely, benign exceptions, contributory exceptions, and page faults. Table 7-5 then shows which combinations of exceptions cause a double fault. The rule is like one of the spelling rules no

Table 7-4
Double-Fault Detection Classes

Class	ID	Description
Benign Exceptions	1	Debug exceptions
	2	NMI
	3	Breakpoint
	4	Overflow
	5	Bounds check
	6	Invalid opcode
	7	Coprocessor not available
	16	Coprocessor error
Contributory Exceptions	0	Divide error
	9	Coprocessor Segment Overrun
	10	Invalid TSS
	11	Segment not present
	12	Stack exception
	13	General protection
Page Faults	14	Page fault

Table 7-5
Double-Fault Definition

		SECOND EXCEPTION		
		Benign Exception	Contributory Exception	Page Fault
FIRST EXCEPTION	Benign Exception	OK	OK	OK
	Contributory Exception	OK	DOUBLE	OK
	Page Fault	OK	DOUBLE	DOUBLE

Table 7-6
Conditions that Cause an Invalid Task State Segment

Error Code	Condition
TSS id + EXT	The limit in the TSS descriptor is less than 103
LTD id + EXT	Invalid LDT selector or LDT not present
SS id + EXT	Stack segment selector is outside table limit
SS id + EXT	Stack segment is not a writable segment
SS id + EXT	Stack segment DPL does not match new CPL
SS id + EXT	Stack segment selector RPL < > CPL
CS id + EXT	Code segment selector is outside table limit
CS id + EXT	Code segment selector does not refer to code segment
CS id + EXT	DPL of non-conforming code segment < > new CPL
CS id + EXT	DPL of conforming code segment > new CPL
DS/ES/FS/GS id + EXT	DS, ES, FS, or GS segment selector is outside table limits
DS/ES/FS/GS id + EXT	DS, ES, FS, or GS is not readable segment

one ever remembers (i before e except after c, etc.). A double fault requires two contributory exceptions, two page faults, or a contributory exception during the execution of the page fault handler.

All double faults cause the processor to push an error code with value 0 onto the stack. The processor cannot restart the faulting instruction. If another exception occurs while the processor is trying to invoke the double fault handler, it simply shuts down completely. Only RESET or a nonmaskable interrupt can revive it at this point.

Double fault handlers should be tasks, not procedures. The context in which the fault occurred may contain a wide variety of problems, such as a lack of stack space or an invalid segment selector. The safest course therefore is to switch tasks, thus guaranteeing the handler a valid context.

INVALID TASK STATE SEGMENT FAULTS

Any condition listed in Table 7-6 can produce an invalid task state segment. The processor pushes an error code onto the stack to help identify the actual cause. The EXT bit indicates whether an outside condition such as an interrupt caused the exception.

This exception occurs during a task switch. If the processor has not completely verified the new TSS, the context is still that of the old task. Otherwise, the context is that of the new task.

One obvious problem here — you must ensure that the handler has a valid TSS. Otherwise, you will end up with a double fault. Thus the handler must be a task invoked via a task gate. After all, you know that the current task state segment is invalid.

SEGMENT NOT-PRESENT EXCEPTIONS

These exceptions occur when the processor tries to use a descriptor with a zero present bit. The program could be loading a data segment register. (Problems with the SS register cause stack exceptions). It could also be loading the LDT register with an LLDT instruction. Or it could be using a gate descriptor that is not present.

This exception is clearly restartable. All that the exception handler must do in most cases is load the segment into memory and set the present bit in its descriptor. The interrupted program can then resume execution by restarting the faulting instruction.

The most common use of this exception is to implement virtual memory at the segment level. However, such implementations are uncommon because the variable size of segments makes them difficult to manage. Instead, most systems implement virtual memory at the page level.

One problem here is that a not-present exception may occur during a task switch. Such a switch assigns new values to all segment registers and may make several of them invalid. For example, the new TSS could have been overwritten or its address could have been specified improperly. Therefore, the exception handler cannot reference memory via the current segment registers, since using them could cause another exception. The result would be a double fault (see Tables 7-4 and 7-5).

How can we escape from this quandary? Either of the following approaches will work:

- Implement the not-present fault handler as a task gate. The processor will load new segment registers from the handler's TSS. However, this approach may make it difficult to determine why the original fault occurred,

since only indirect access to the suspended routine's state is possible. The handler must use the back link in its TSS to gain access.

- Check all segment register images in the TSS, simulating the usual processor test. You can also force tests by PUSHing and POPping all segment registers. This approach involves some work but allows the exception handler to run in the interrupted task's context.

Segment not-present exceptions may have special significance if their cause is a not-present bit in a gate descriptor. Systems software may use that indicator for special functions such as network transfers or the emulation of features that are under development.

STACK EXCEPTIONS

Two general conditions cause stack exceptions:

- Limit violations in operations that use the SS register. Such operations include POP, PUSH, ENTER, and LEAVE, as well as instructions that use SS implicitly (by addressing through ESP or EBP) or explicitly (through a segment override). Typical examples of such instructions are MOV ECX,[EBP] and MOV EAX,SS:[ESI+7].
- An attempt to load SS with a not-present descriptor. Note that this results in a stack exception, not a segment not-present exception.

Stack exceptions cause the processor to push an error code onto the exception handler's stack. The code is usually zero. It contains a selector to a segment if a not-present stack segment caused the exception or if an interlevel CALL caused the new stack to overflow.

Stack exceptions are always restartable. Note that, in the case of a not-present descriptor, the instruction to be restarted is the first instruction of the new task.

The same problem with regard to task switches applies here as with segment not-present exceptions. Other segment registers may not contain valid values. The solutions described in the previous section apply here also.

GENERAL PROTECTION EXCEPTIONS

This category is the grab bag for all protection violations that don't fit anywhere else. Among the possibilities are:

- Exceeding the limits of segments or descriptor tables. Note, however, that limit violations involving the SS register cause stack exceptions.
- Violating the accessibility of a segment. This sounds either dangerous or disgusting. Typical examples are transferring control to a segment that is not executable, writing into one that is executable or read-only, reading from one that is executable, and loading a data segment register with a descriptor for a system or unreadable segment.
- Trying to use a null selector.
- Switching to a busy task.
- Violating privilege rules. Also trying to exit V86 mode via a trap or interrupt gate to a nonzero privilege level.
- Trying to enter an impossible state, such as one that has paging enabled ($PG = 1$) without protection ($PE = 0$).

In response to a general protection exception, the processor pushes an error code onto the handler's stack. The error code is zero unless loading a descriptor caused the exception. In that case, the error code contains a selector to the descriptor. This selector may refer to an instruction operand, a gate, or a task state segment.

You can avoid many protection exceptions or clarify their causes by testing segment selectors ahead of time. The VERR (verify for reading), VERW (verify for writing), LAR (load access rights), and LSL (load segment limit) instructions are useful for this purpose. VERR, VERW, LAR, and LSL are not privileged.

PAGE FAULTS

This exception usually means that an accessed page is not currently in memory and must be loaded from disk. It is the key to demand-paged virtual memory systems. Another possibility is that the page is present but the procedure is not privileged enough to access it.

Besides saving a special error code (see Figure 7-5) in the stack, the processor also saves the linear address that caused the exception. This address ends up in control register 2. The exception handler needs it to locate the page directory and page table entries.

One problem is that page faults may occur during task switches. These are difficult to handle because they may occur before or after the change of context or before the new context has been verified. The only way to ensure a valid context for the page fault handler is to invoke it via a task gate.

Another problem is that a page fault may occur with an invalid stack pointer. The usual cause is a page fault in the middle of the 8086 sequence:

```
MOV SS,AX
MOV SP,STACKTOP
```

The result is a mixture of new stack segment and old stack pointer. The solution is to use the LSS (load full stack pointer) instruction instead of the 8086 sequence. LSS loads the stack segment and stack pointer registers as a unit.

Page faults may have special meanings to operating systems. For example, an OS may use them to load programs, regain control, extend the stack, or copy data pages for new programs.

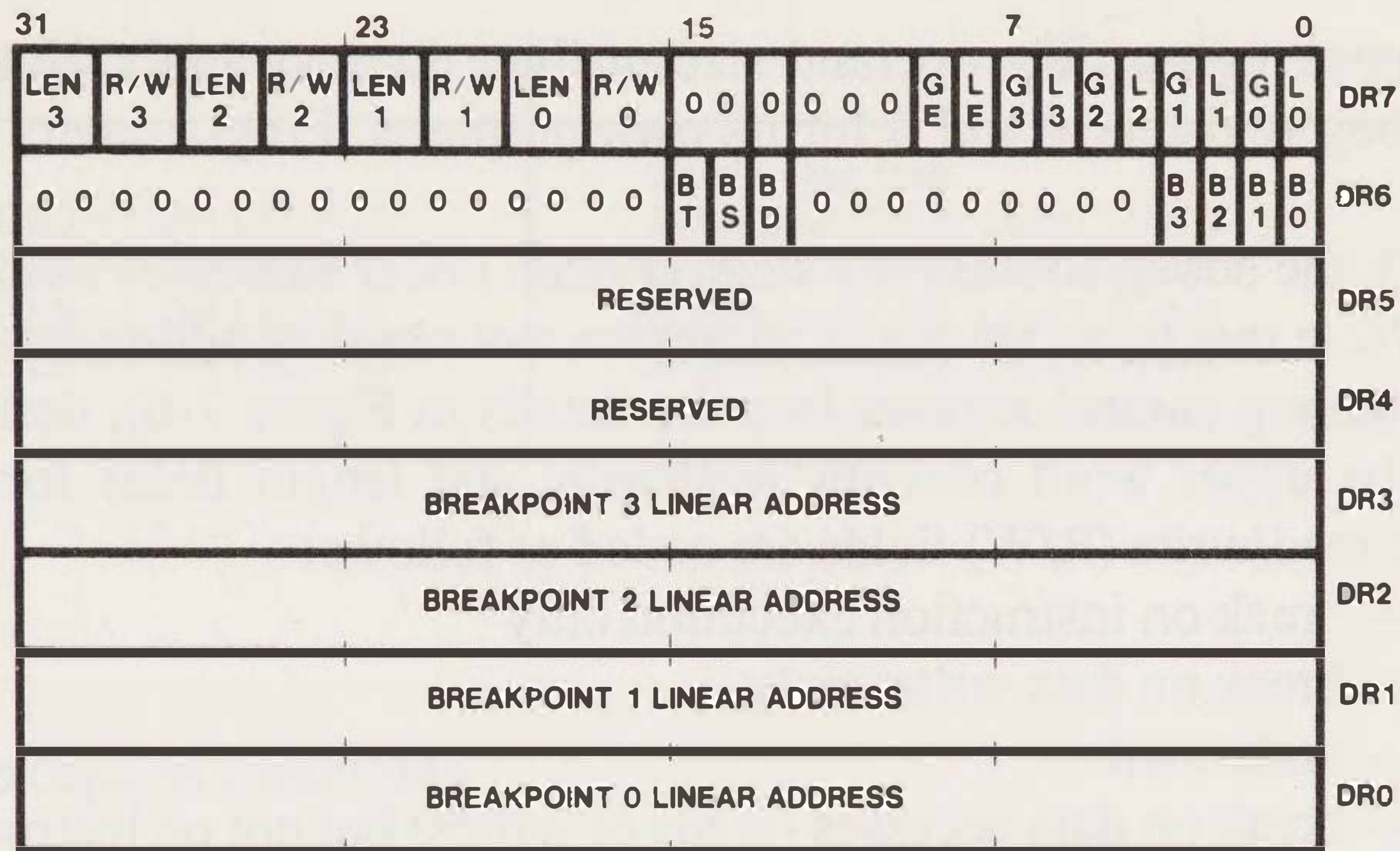
DEBUGGING FEATURES

One major use of exceptions is in debugging. INT 3 (the 1-byte instruction breakpoint) is particularly valuable here. Debuggers can easily replace the first byte of an instruction with INT 3, thus causing a trap back to a monitor or other systems software. This kind of breakpoint has been common in processors for many years.

What are its limitations? First, it can produce only instruction breakpoints. You cannot use it to cause breakpoints on the reading or writing of specific data addresses. Second, replacement is impossible when an instruction is in ROM and often difficult when it is in the operating system. Besides, the debugger must physically replace an instruction, thus changing the underlying program. It must also restore the instruction later. One common problem in debugging is leftover breakpoints resulting from runaway programs or improper termination of the debugger.

The 80386 provides special hardware debugging features to overcome these limitations. They are:

- A reserved debug interrupt vector (exception 1).
- Four debug address registers that programmers can use to monitor addresses. No instruction replacement is necessary, so these addresses can be anywhere. They can be in either data memory or program memory.
- A debug control register that programmers can use to specify debug conditions.
- A debug status register that helps identify the cause of a debug exception.



NOTE: 0 MEANS INTEL INTEL RESERVED. DO NOT DEFINE.

Figure 7-6

80386 debug registers.

- The trap (T) bit of a task state segment to allow monitoring of task switches. T is bit 0 of the highest-addressed double word in the standard TSS (see Figure 6-1).
- The resume flag (RF) in the flags register. It allows instruction restart after a debug exception for testing purposes. RF is bit 16 of the extended flags (see Figure 2-3).
- The single-step (trap) flag (TF) in the flags register which forces a debug exception after every instruction. TF is bit 8 of the extended flags (see Figure 2-3).

These features make it easy to set breakpoints on such conditions as the following:

- Task switch to a specific task
- Instruction execution or data read or write at a specified address

DEBUG REGISTERS

The 80386 has eight debug registers, as shown in Figure 7-6. You can access most of them through special MOV instructions that must execute at privilege level 0 (the most privileged level). MOV allows only 32-bit transfers between a debug register and a

general-purpose register. The operand-size attribute does not apply. Note that Intel has reserved debug registers 4 and 5 for its own purposes. Programmers cannot access them.

DR0-DR3, the debug address registers, contain linear addresses used in breakpoint conditions. Note that these are linear addresses, not physical addresses.

DR7, the debug control register (see the details in Figure 7-6), defines the debug conditions. Its upper word contains read/write and length fields for each address register. The read/write (R/W) fields are coded as follows:

- 00 — break on instruction execution only
- 01 — break on data writes only
- 10 — undefined
- 11 — break on data accesses (reads or writes) but not on instruction fetches

The length (LEN) fields have the following meanings:

- 00 — 1-byte length
- 01 — 2-byte length
- 10 — undefined
- 11 — 4-byte length

This information applies only to data transfers. The field should be 00 for instruction fetches; all other values are undefined.

The lower word of DR7 selectively enables the four address breakpoint conditions. Local (L) enables refer only to the current task; the processor resets them at every task switch. Global (G) enables refer to all tasks; task switches do not affect them. The LE and GE bits cause the reporting of a data breakpoint on the instruction that causes it. One of these bits should be set to use data breakpoints.

DR6, the debug status register (see the details in Figure 7-6), contains the following bits:

B0 through B3 indicate whether a debug exception has occurred under the conditions for a specific address register. These bits are set if an exception has occurred, regardless of whether it was enabled with a G or an L bit.

BT is set if a task switch has occurred and the T (trap) bit of the new TSS is set.

BS (great name!) is set if a single-step exception has occurred. This is the highest-priority debug exception.

BD is set if the next instruction will read or write a debug register while Intel's ICE-386 (a popular hardware debugging device) is using the debug registers.

Note that the processor sets the bits in the status register but never clears them. The debug handler must clear them explicitly.

80386 Exceptions and Debugging Features

Flags to Test	Condition
BS=1	Single-step trap
B0=1 AND (GE0=1 OR LE0=1)	Breakpoint DR0, LEN0, R/W0
B1=1 AND (GE1=1 OR LE1=1)	Breakpoint DR1, LEN1, R/W1
B2=1 AND (GE2=1 OR LE2=1)	Breakpoint DR2, LEN2, R/W2
B3=1 AND (GE3=1 OR LE3=1)	Breakpoint DR3, LEN3, R/W3
BD=1	Debug registers not available; in use by ICE-386.
BT=1	Task switch

Figure 7-7

80386 debug exception conditions.

DEBUG EXCEPTIONS

The 80386 reserves interrupt 1 for debug exceptions. Table 7-7 lists the possible causes. The debugger can differentiate among these by examining the debug control and status registers (DR7 and DR6, respectively). Instruction address breakpoints are faults, whereas other debug conditions are traps.

SUMMARY

Exceptions are internal conditions or instructions that cause the 80386 to suspend its normal activities and do special routines. The 80386 handles them just as it does external interrupts. It transfers control to a new routine through an entry in the interrupt descriptor table (IDT). The entry may be an interrupt, trap, or task gate. Interrupt and trap gates differ only in their effect on the Interrupt Enable (I) flag; interrupt gates clear the flag, disabling interrupts, whereas trap gates leave it unchanged. A task gate provides a completely new context for the exception handler.

Exceptions may be either processor detected or programmed. Programmed exceptions are the so-called software interrupts INTO, INT 3, INT n, and BOUND. Processor-detected exceptions include divide errors, debug exceptions, invalid opcodes, coprocessor errors, invalid task state segments, segment not-present errors, stack exceptions, page faults, and general protection exceptions.

Exceptions can be classified (in order of seriousness) as faults, traps, or aborts. The processor can restart a faulted instruction. It can go on to the next instruction after ser-

ving a trap. It can only report the error in the case of an abort. Double faults occur when there is a serious exception during the execution of the handler for a serious exception. If still another fault occurs during the execution of the double fault handler, the processor shuts down completely.

Faults that occur during task switches cause special problems. The major difficulty is that the processor may not have checked the new task's state completely. Thus more faults may occur if the processor tries to use segment registers and other resources. The usual solution is to implement the fault handler as a task with its own (known valid) context.

The 80386 provides special debugging facilities as well as the usual traps, breakpoints, and single-step mode. These facilities allow the programmer to set breakpoints at any of four addresses under a variety of conditions. No slowdown occurs, and no instructions need to be replaced. Breakpoints may occur on instruction fetches, data accesses, or data writes.

80386 Hardware Features

Only a signal shown and a distant voice in the darkness.
Longfellow, *Tales of a Wayside Inn*

Handshakes can be faked and usually are, but smiles can't.
Rex Stout, *Homicide Trinity*

This chapter covers the 80386's hardware features. It describes the processor's signal structure, bus operations, memory interface, and numeric coprocessors. A final section discusses cache memory.

NEW 80386 FEATURES

The 80386's new hardware features are:

- Full 32-bit address and data buses with automatic handling of byte, word, and misaligned transfers. Intel calls this feature *dynamic data bus sizing*.
- Address pipelining that allows the overlapping of successive memory cycles. It gives extra time for a memory access without reducing overall system performance.

- Provisions for both 32-bit and 16-bit data buses
- Support for a 32-bit numerical coprocessor, the 80387 (and for the 32-bit Weitek 1167 floating point chip set).

The 80386's high clock speeds (16 MHz and up) mean that it needs fast memory to operate without delays. However, a large amount of such memory is expensive. Ways to reduce costs without sacrificing performance include:

- Overlapping bus cycles so that the next memory or I/O access can begin before the current one ends.
- Dividing memory into banks (units with their own control circuitry) so that accesses to successive addresses can occur without delays. A single bank may require waiting time between accesses.
- Saving frequently used instructions and data in a small amount of high-speed memory (called a *cache*). Cache memory has the same relationship to main memory that main memory has to disk storage.

All three techniques have been used previously in the design of larger computers.

80386 EXTERNAL SIGNALS

Table 8-1 lists the 80386's signal pins, along with their functions and characteristics. A # symbol after a signal's name indicates that it is active low (0 is the named or active state; 1 is the opposite or inactive state). The terms commonly used to identify states are *asserted* (that is, in the active state) and *negated* (that is, in the inactive state). If the signal has two names, the one followed by # is the low (0) state. We may group the status and control signals (not including the data bus, address bus, and clock) into the following categories:

- Memory and I/O transfer (handshake)
- Startup
- Coprocessor
- Interrupt
- DMA

Memory and I/O Transfer Control Signals

The memory and I/O transfer control signals are described in the following paragraphs.

Table 8-1
Summary of 80386 signal pins

Signal Name	Signal Function	Active State	Input/Output	Input Synch or Asynch to CLK2	Output High Impedance During HLDA?
CLK2	Clock	—	I	—	—
D0-D31	Data Bus	High	I/O	S	Yes
BE0#-BE3#	Byte Enables	Low	O	—	Yes
A2-A31	Address Bus	High	O	—	Yes
W/R#	Write-Read Indication	High	O	—	Yes
D/C#	Data-Control Indication	High	O	—	Yes
M/IO#	Memory-I/O Indication	High	O	—	Yes
LOCK#	Bus Lock Indication	Low	O	—	Yes
ADS#	Address Status	Low	O	—	Yes
NA#	Next Address Request	Low	I	S	—
BS16#	Bus Size 16	Low	I	S	—
READY#	Transfer Acknowledge	Low	I	S	—
HOLD	Bus Hold Request	High	I	S	—
HLDA	Bus Hold Acknowledge	High	O	—	No
PEREQ	Coprocessor Request	High	I	A	—
BUSY#	Coprocessor Busy	Low	I	A	—
ERROR#	Coprocessor Error	Low	I	A	—
INTR	Maskable Interrupt Request	High	I	A	—
NMI	Non-Maskable Intrpt Request	High	I	A	—
RESET	Reset	High	I	S	—

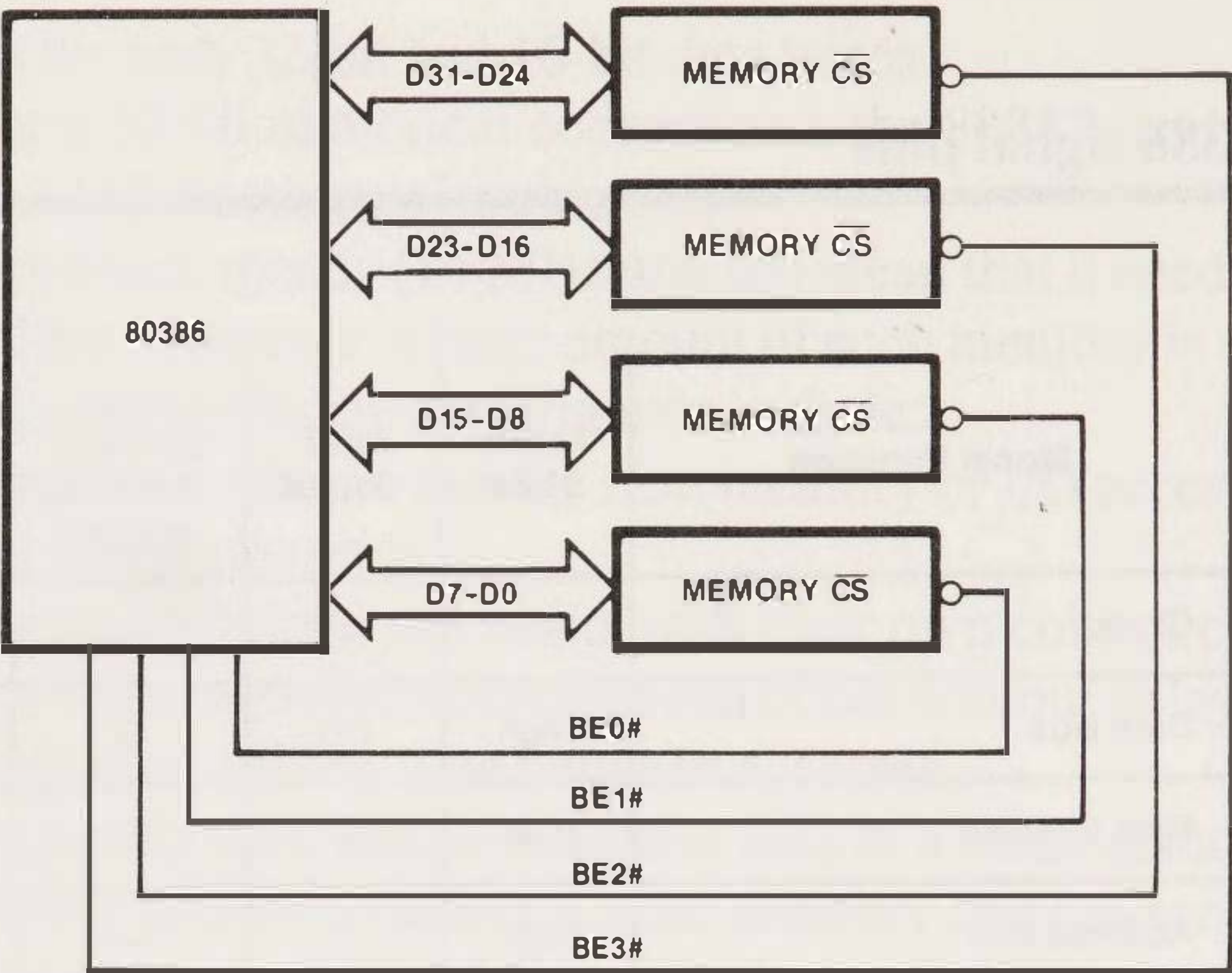


Figure 8-1
Using the byte enable (BE) signals to select 8-bit banks of memory.

Table 8-2
Possible Data Transfers on the 32-Bit Data Bus

Possible Data Transfers to 32-Bit Memory	
Size	Byte Enables
32 bits	3-2-1-0
24 bits	3-2-1 2-1-0
16 bits	3-2 2-1 1-0
8 bits	3 2 1 0

	BYTE ADDRESS	WORD ADDRESS	DWORD ADDRESS
BE0	0	0	0
BE1	1	0	0
BE2	2	2	0
BE3	3	2	0
BE0	4	4	4
BE1	5	4	4
BE2	6	6	4
BE3	7	6	4
BE0	8	8	8
—	—	—	—
—	—	—	—
—	—	—	—

31	24	23	16	15	8	7	0
BE3#		BE2#		BE1#		BE0#	

Figure 8-2

Address, data bus, and byte enables for 32-bit bus.

BE0# through BE3# are enable signals that select which bytes the processor will transfer during an I/O or memory cycle. Computer designs generally use the BE signals to select 8-bit units (banks) of memory as shown in Figure 8-1. BE0# enables transfers over data bus lines 0 through 7 (D0 through D7), BE1# over D8 through D15, BE2# over D16 through D23, and BE3# over D24 through D31. Figure 8-2 shows the correspondences among byte enables, byte addresses, word addresses, and double word addresses. Assuming the design in Figure 8-1, the processor can transfer a double word by asserting all four enables. In fact, it can transfer any contiguous set of bytes. All it must do is assert the enables given by an entry from Table 8-2.

The processor can also use the BE signals to transfer misaligned words or double words. That is, those not starting at an address divisible by 2 or 4. Such transfers require two memory cycles instead of one. Table 8-3 lists the contents of the address bus and the active byte enables during both cycles for all alignments. Figure 8-3 shows the cycles from the memory's point of view. The example is a 32-bit transfer with an even but misaligned address (that is, an address divisible by 2 but not by 4). Note that the 24-bit transfers listed in Table 8-2 occur only as part of the transfer of a misaligned double word. There are no 24-bit instructions.

Although byte enable signals are convenient for local memory control, they may not be adequate for multiprocessor or external bus-based systems. Standard buses such as Multibus I, Multibus II, and VME generally require explicit A0 and A1 signals. Extra

Table 8-3
Misaligned Data Transfers on a 32-Bit Bus

Transfer Type	Physical Address	First Cycle:		Second Cycle:	
		Address Bus	Byte Enables	Address Bus	Byte Enables
Word	$4N + 3$	$4N + 4$	0	$4N$	3
Doubleword	$4N + 1$	$4N + 4$	0	$4N$	1-3
Doubleword	$4N + 2$	$4N + 4$	0-1	$4N$	2-3
Doubleword	$4N + 3$	$4N + 4$	0-2	$4N$	3

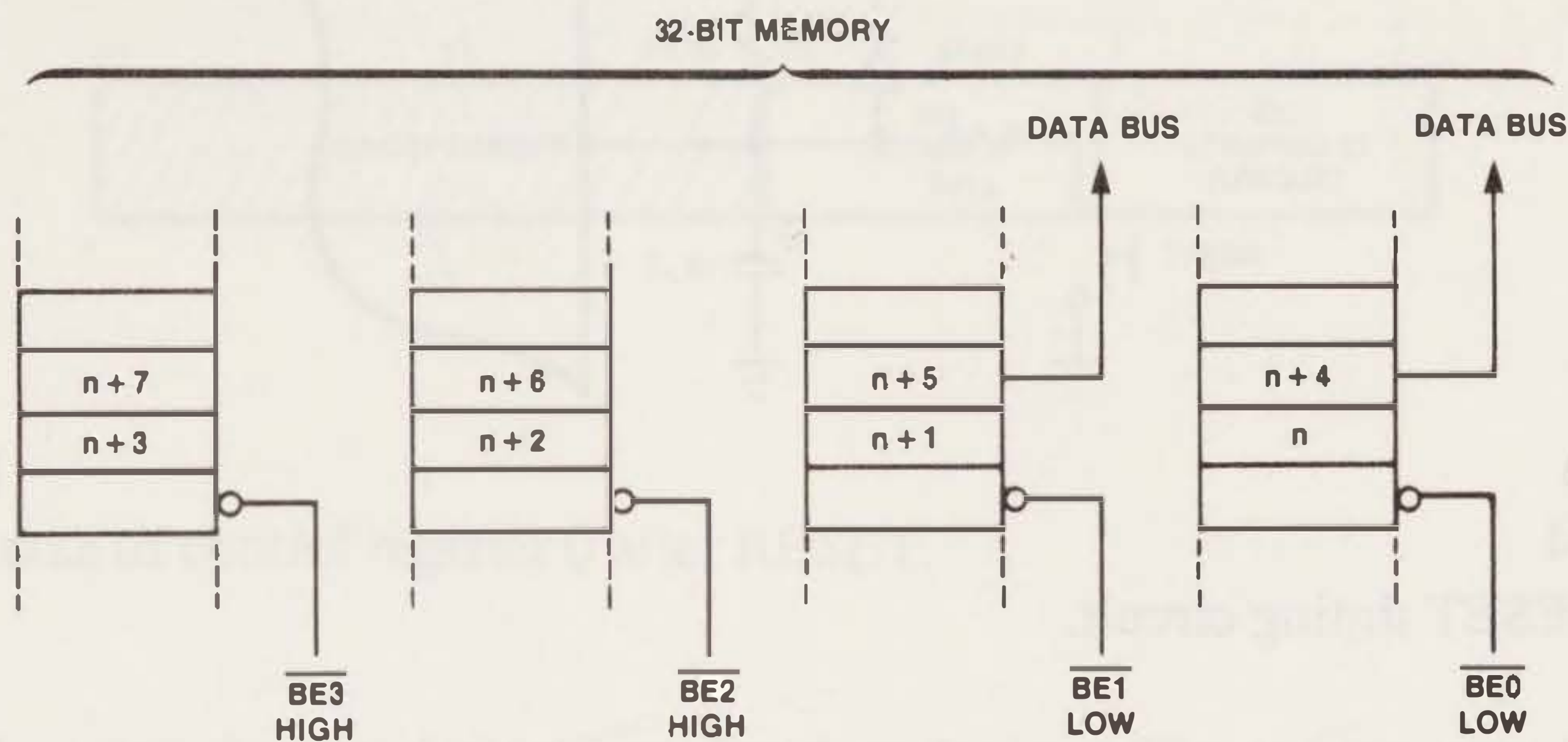
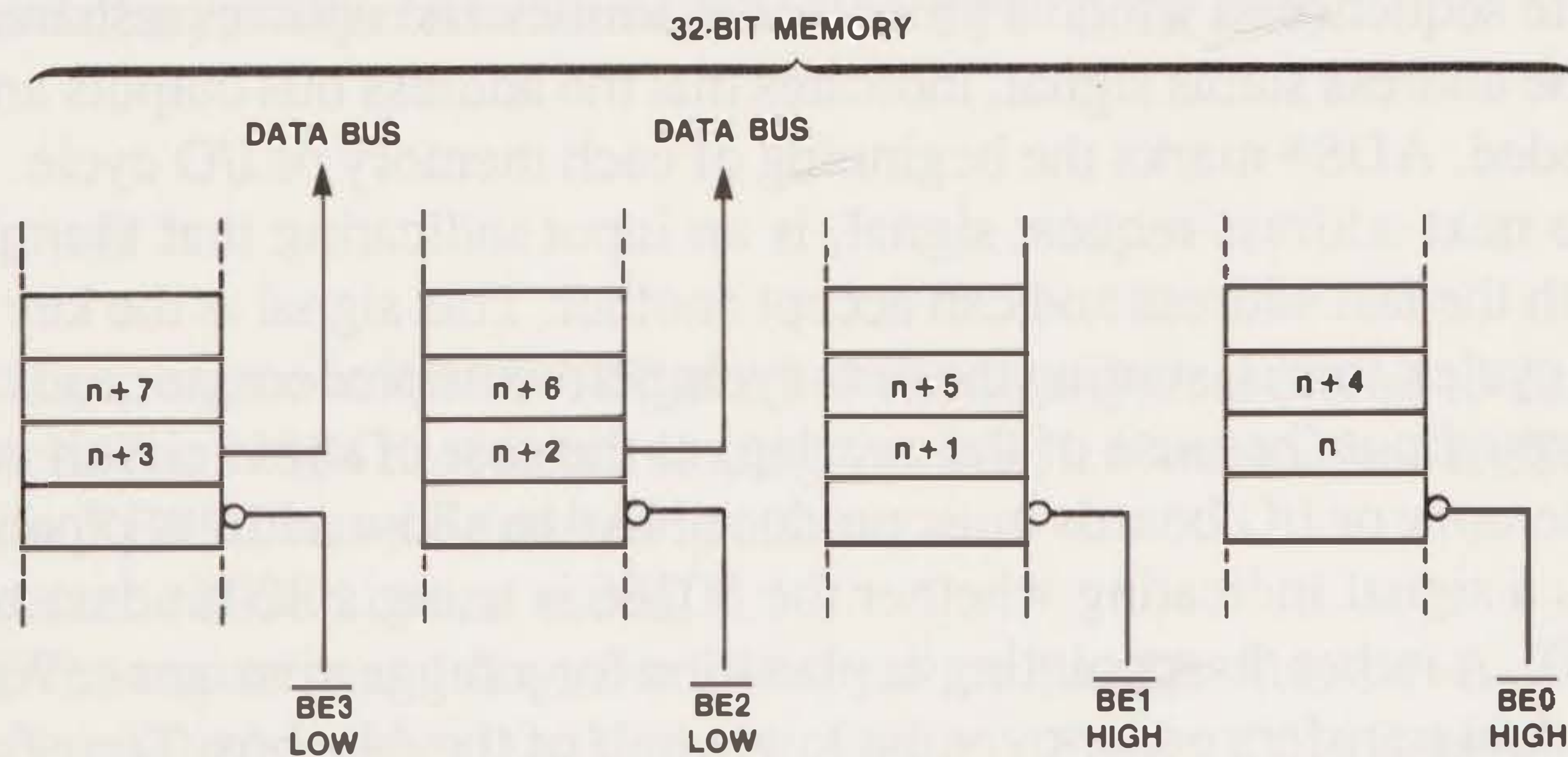
NOTE: $4N$ = Nth doubleword address

gates are necessary to produce them. For descriptions of these buses, see the documentation from Intel and the Multibus Manufacturers' Group for Multibus and from Motorola and the VME Users' Group for the VME bus.

W/R# is a write-read indicator. It differentiates between input (0) and output (1) cycles.

D/C# is a data-control indicator. It differentiates between cycles used for control purposes (0) and those used to transfer data (1). Memory and I/O devices may use the bus only during data cycles. Other devices may use the bus during control cycles. For example, interrupt acknowledge cycles send D/C# low. The result is to activate neither memory nor I/O. The vector source can then put the interrupt type on the data bus without having to contend with memories or input ports.

M/I/O# is a memory-I/O indicator. It differentiates between cycles used to access memory (1) and those used to access I/O (0). This signal is the only difference between input/output (IN, OUT) and memory transfer (MOV) instructions. As far as the 80386 is concerned, an I/O device is anything activated by this signal being low. A memory is anything activated by it being high. All other distinctions (size, function, speed, etc.) are beyond the 80386's comprehension. Note that it is thus perfectly acceptable for the 80386 to access I/O devices through memory addresses (called *memory-mapped I/O*) or memory through I/O addresses (perhaps a local buffer for an I/O device).

FIRST BUS CYCLE: $A_{31} - A_2 = n + 4$ SECOND BUS CYCLE: $A_{31} - A_2 = n$ **Figure 8-3**

A 32-bit misaligned data transfer.

LOCK#, the bus lock indicator, is used to identify special cycles in multiprocessor systems. This signal generally tells other processors that they may not take control of the address and data buses. The usual reason is that the processor is doing an indivisible operation such as updating a selector and an offset. If another processor took control

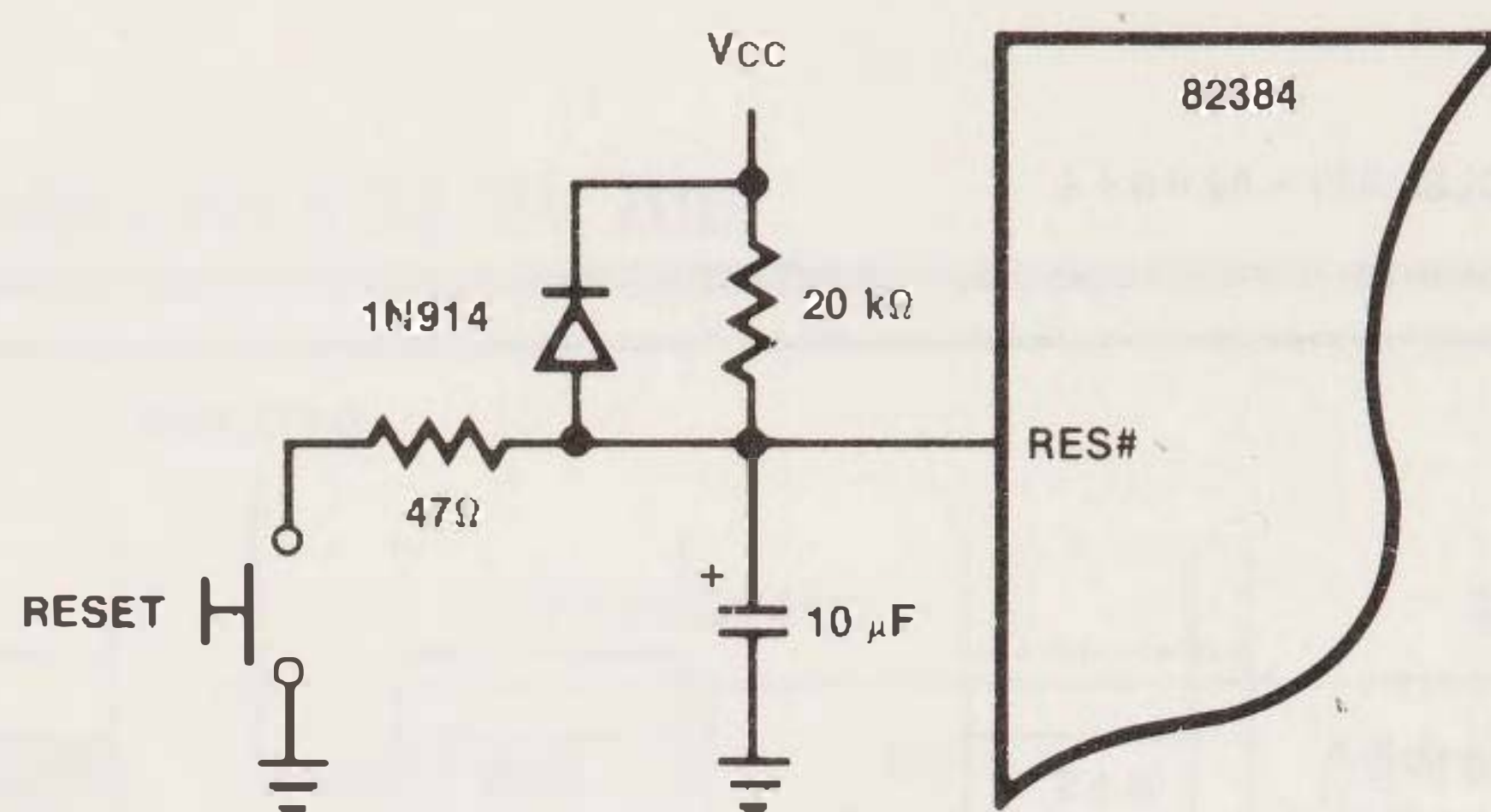


Figure 8-4
Typical RESET timing circuit.

during the operation, it might find only one of the address' two parts updated. A processor may also assert LOCK# while changing shared memory, control flags (*semaphores*), or interface parameters. Note that it may be necessary to lock a read-modify-write sequence in which a processor examines and updates a shared location.

ADS#, the address status signal, indicates that the address bus outputs are valid and can be decoded. ADS# marks the beginning of each memory or I/O cycle.

NA#, the next address request signal, is an input indicating that the memory has finished with the last address and can accept another. This signal is the key to pipelining address cycles, that is, starting the next cycle before its predecessor ends. The result is higher throughput (because of the overlap) at the cost of some circuit complexity. Note that memory or I/O boards must produce NA# to allow address pipelining.

BS16# is a signal indicating whether the 80386 is using a 32-bit data bus (1) or a 16-bit bus (0). A rather disappointing explanation for a suggestive name. When BS16# is asserted, data transfers occur over the lower half of the data bus. Transfers of more than 16 bits take two cycles.

BS16# is an input signal that can help connect the 32-bit processor to a 16-bit memory section. Presumably, this makes sense only as a stopgap measure for 16-bit systems. Connecting 32-bit processors to 16-bit buses increases throughput over strictly 16-bit systems. For example, many companies make add-on boards that put an 80386 CPU in an 80286-based computer such as an IBM PC AT. Of course, the overall system data bus remains 16 bits wide. Ultimately, however, the situation is like having a brand new 8-lane bridge with 4-lane connector roads at both ends.

READY# is an acknowledgment from the memory or I/O section, indicating the successful completion of a transfer. Circuitry on a memory or I/O board can extend

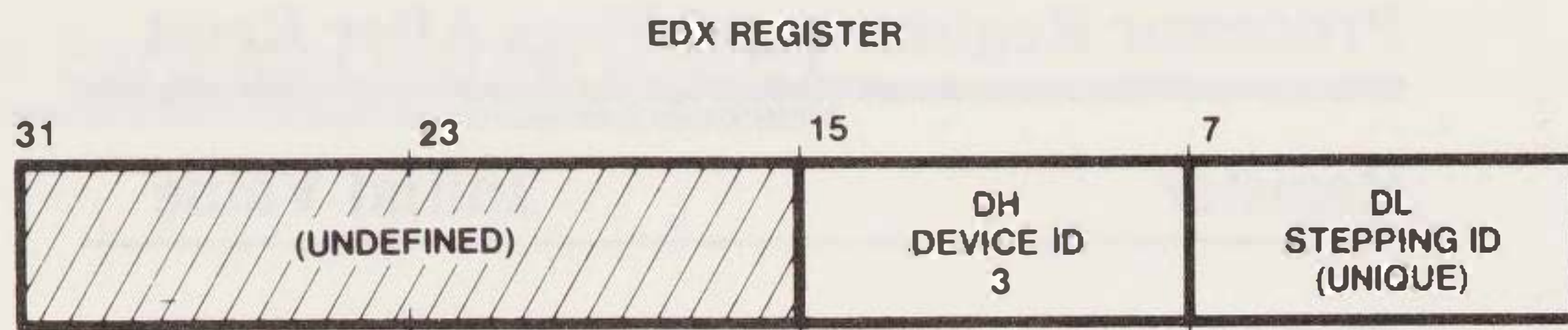


Figure 8-5

Initial contents of control register 0 after RESET.

READY# by extra clock periods to wait for slow devices. The common use is for slow memory. That memory could be cheaper, use less power, resist radiation damage, retain its contents indefinitely, or have other special properties. READY# can also be used to interface fast I/O devices such as disks and image processing boards that run at close to CPU speeds (say, within a factor of 10 of the CPU clock rate).

Startup Signals

The main startup signal is RESET. Figure 8-4 shows a typical circuit to derive it from a panel switch. The 82384 device is a clock generator; it synchronizes RESET with the system clock. RESET has the following effects:

- It puts the 80386 in real mode.
- It gives registers and flags the initial values listed in Table 8-4.
- It sets control register 0 as shown in Figure 8-5. The ET (extension type, not extraterrestrial) bit indicates whether the system has an 80387 coprocessor (1) or not (0).
- It puts the output pins in the states listed in Table 8-5. Note that the address lines are all high and the byte enables are all active.

Note that, after RESET, the processor automatically brings address lines A20 through A31 high during instruction fetches. Thus instruction execution begins at physical address FFFFFFF0H, not FFF0H, as the values of CS and IP alone would suggest. The first far (intersegment) JMP or CALL brings address lines A20 through A31 low so that the processor continues executing instructions in the bottom 1 Mb of physical memory. RESET is usually applied separately to other system devices, such as

Table 8-4
Processor Registers and Flags After Reset

Register	Initial Value
CR0	See Figure 8-5
CS selector	0
DS selector	0
DX	Device ID (Figure 8-6)
EAX	Depends on self-test if requested (0 means “passed”).
EFLAGS	2 (Parity flag = 1)
EIP	FFF0H
ES	0
FS	0
IDTR base	0
IDTR limit	3FFH
SS	0
All other registers	Undefined

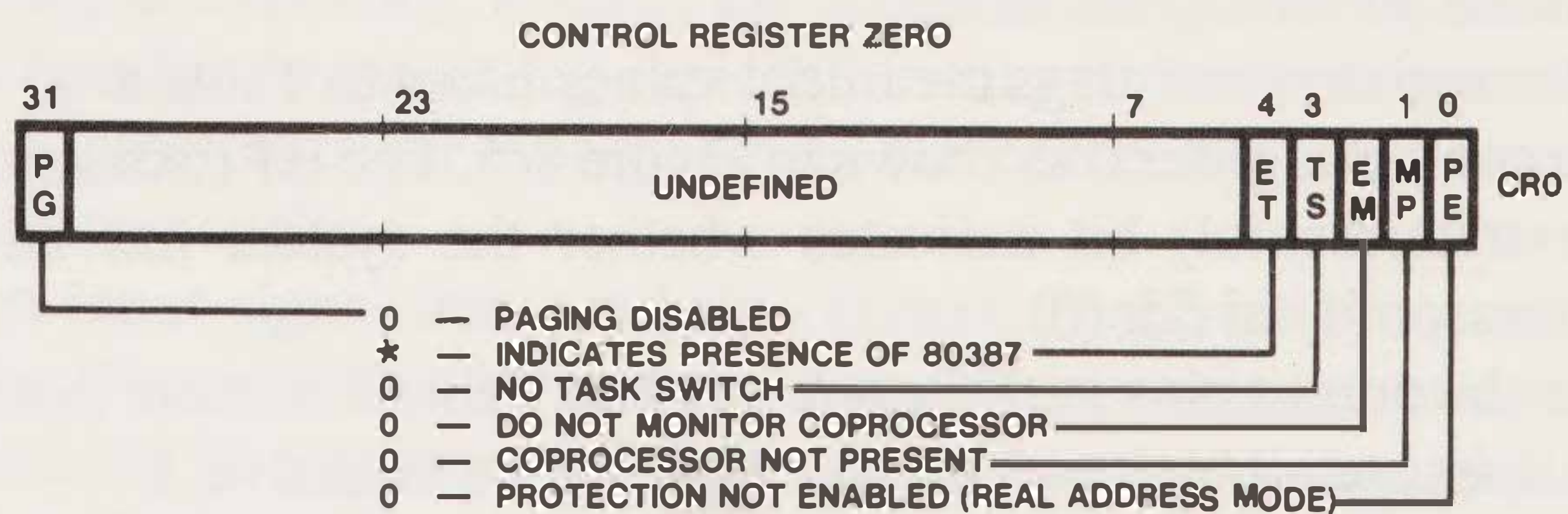


Figure 8-6
 Contents of EDX register after RESET.

Table 8-5
80386 Output Pin States During RESET

Pin Name	Pin State
LOCK#, D/C#, ADS#, A31-A2	High
W/R#, M/IO#, HLDA, BE3#-BE0#	Low
D31-D0	Three-State

coprocessors (80287 or 80387), parallel interfaces (8255), and DMA controllers (82258 or 82380).

Coprocessor Signals

The coprocessor signals are:

- **BUSY#** is a status signal from the coprocessor indicating that it is not done with its current operation. An active **BUSY#** means that the coprocessor cannot accept a new instruction (the 80387 allows overlap in some cases).
- **ERROR#** is a status signal from the coprocessor indicating that its latest operation produced an error. Typical causes are an invalid operation, overflow, a zero divisor, underflow, a denormalized operand (outside the device's range), or an inexact result. An invalid operation could be stack overflow or underflow or the encountering of an indefinite form such as 0/0. During initialization, **ERROR#** indicates whether the system has an 80387 coprocessor.
- **PEREQ** (coprocessor request) is a status signal from the coprocessor indicating that it is ready to transfer data. The coprocessor does not control the address and data buses on its own. Instead, it depends on the 80386 for all data transfers.

Interrupt Control Signals

The interrupt inputs are:

- INTR, the maskable interrupt used for I/O and other normal system functions. The input is level sensitive.
- NMI, the nonmaskable interrupt used for catastrophic events such as power failure or bus parity errors. The input is edge sensitive so that it will not interrupt its own service routine.

There is no interrupt acknowledge output. Instead, the processor responds to INTR interrupts with special interrupt acknowledge cycles. We will discuss them when we describe bus operations. NMI interrupts have their own fixed vector (#2; see Table 4-3), so they do not need acknowledge cycles.

DMA Signals

The DMA signals are:

- A HOLD input that tells the 80386 to relinquish its buses to the external controller (*bus master*).
- An HLDA (hold acknowledge) output that informs the external bus master that it may take control of the bus.

A DMA controller (usually a single chip with some peripheral circuitry) must manage the DMA system. It must provide handshaking and control the activation and prioritization of individual DMA channels. The requestor must keep HOLD active as long as it needs the bus. It must not take control of the bus before receiving the HLDA acknowledgment from the processor.

80386 BUS OPERATION

Figure 8-7 contains timing diagrams for nonpipelined read cycles. The cycle at the left operates at full speed, whereas the one at the right includes an extra wait state. Figure 8-8 contains similar diagrams for write cycles. A cycle consists of at least two bus states, designated as T1 and T2. Each bus state in turn consists of two CLK2 cycles (CLK2 runs at twice the internal processor clock frequency). Some diagrams designate the CLK2 cycles as ϕ_1 and ϕ_2 , respectively.

Nonpipelined 80386 bus cycles work as follows:

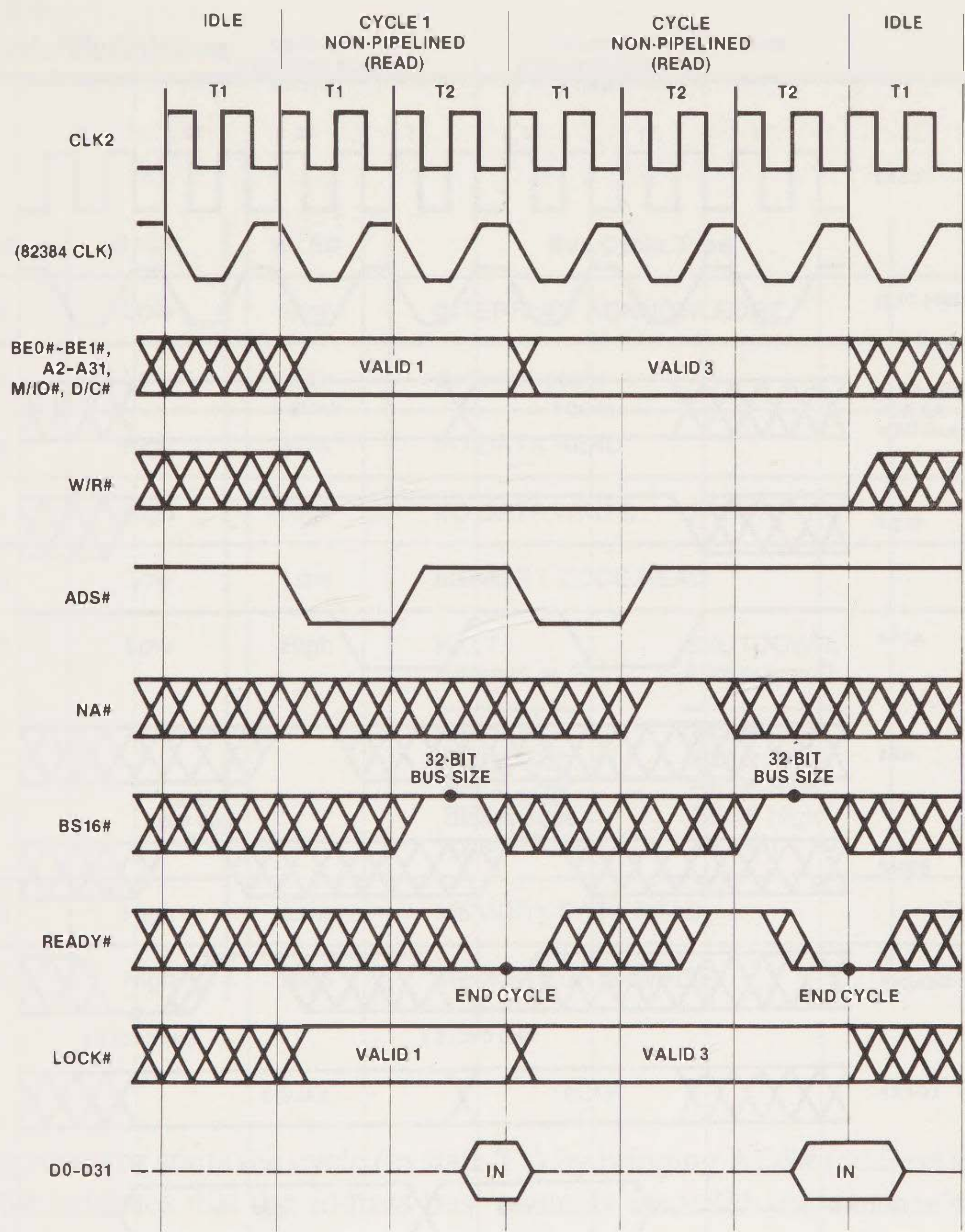


Figure 8-7
Non-pipelined address read cycles.

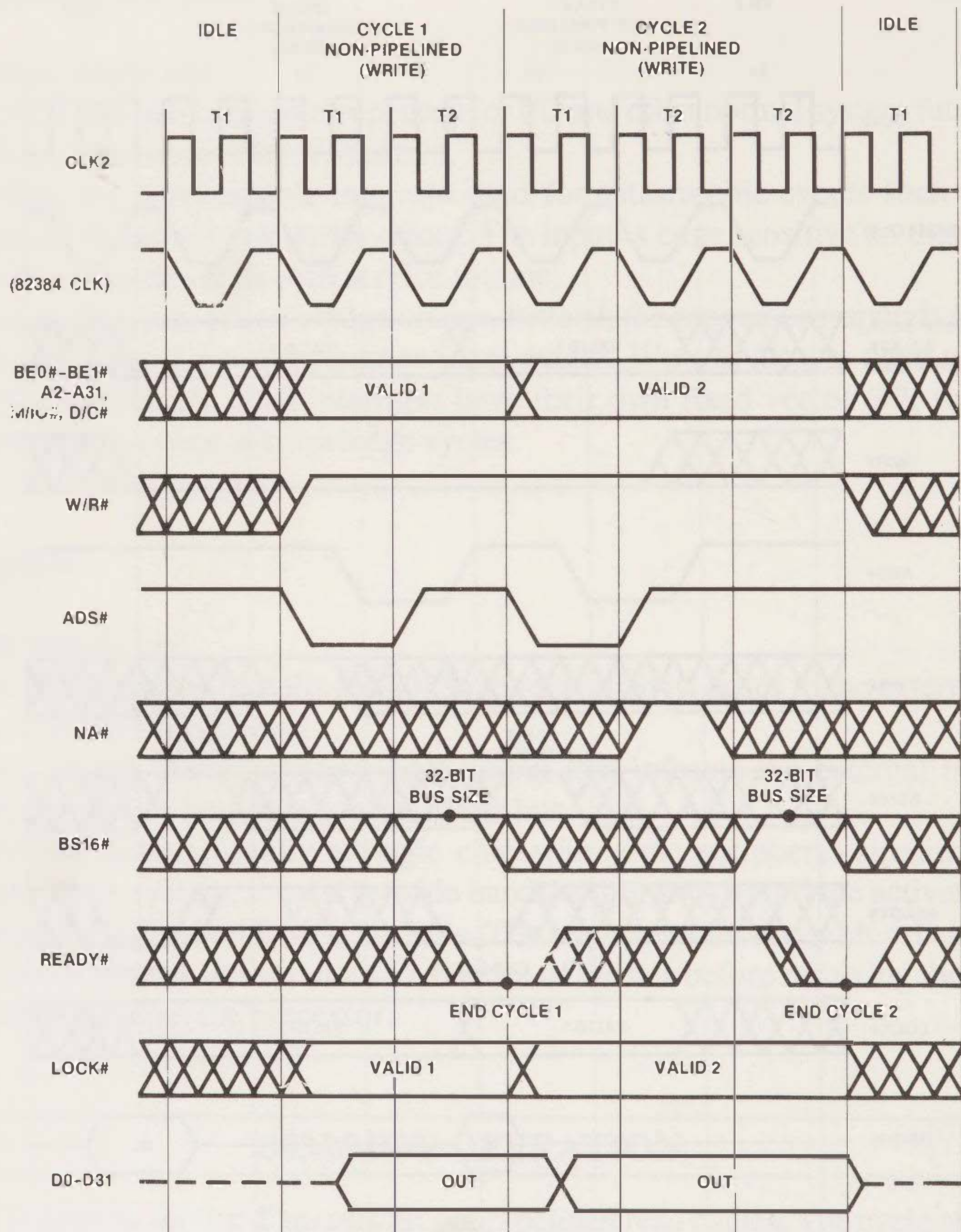
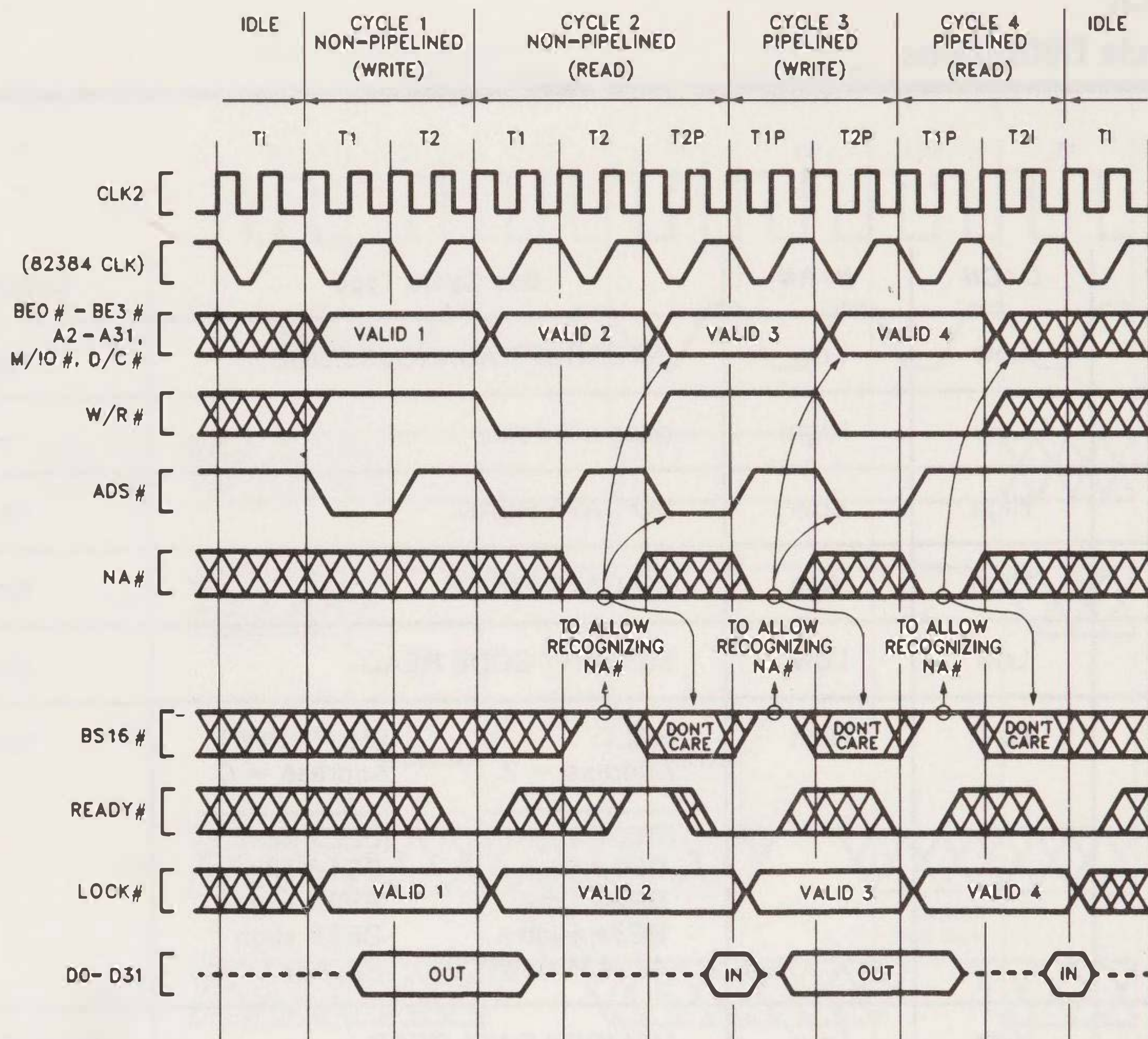


Figure 8-8
Non-pipelined address write cycles.

Table 8-6
Bus Cycle Definitions

M/IO#	D/C#	W/R#	Bus Cycle Type	Locked?
Low	Low	Low	INTERRUPT ACKNOWLEDGE	Yes
Low	Low	High	does not occur	—
Low	High	Low	I/O DATA READ	No
Low	High	High	I/O DATA WRITE	No
High	Low	Low	MEMORY CODE READ	No
High	Low	High	<div> <div> HALT: Address = 2 </div> <div> SHUTDOWN: Address = 0 </div> </div> <div> <div> (BE0# High BE1# High BE2# Low BE3# High A2-A31 Low) </div> <div> (BE0# Low BE1# High BE2# High BE3# High A2-A31 Low) </div> </div>	No
High	High	Low	MEMORY DATA READ	Some Cycles
High	High	High	MEMORY DATA WRITE	Some Cycles

1. The processor starts the cycle (in state T1) by bringing ADS# (address status) low. ADS# indicates that the address bus' contents are valid and can be decoded and latched.
2. The processor brings the memory control signals to the states appropriate for the current cycle. These signals include the byte enables (BE3# through BE0#) and the bus status outputs (M/IO#, D/C#, W/R#, and LOCK#). Table 8-6 defines different types of bus cycles in terms of these signals. Obviously, memory read cycles are by far the most common type, as all instruction fetches fall in this category.



Following any idle bus state (Ti), addresses are non-pipelined. Within non-pipelined bus cycles, NA# is only sampled during wait states. Therefore, to begin address pipelining during a group of non-pipelined bus cycles requires a non-pipelined cycle with at least one wait state (Cycle 2 above).

Figure 8-9
Pipelined address cycles.

- At the end of T2, the processor samples READY#. If it is active (low), the processor reads the input data in a read cycle or terminates a write cycle. If READY# is inactive, the processor waits for another clock cycle (designated as an extra T2 in the extended cycles of Figures 8-7 and 8-8) before sampling it again.

The extension process continues as long as READY# remains inactive. This creates an obvious problem if a programming error makes the processor access a nonexistent address. The philosophical rule (very Zen-like) is that nonexistent memory is never ready. The usual solution is to have a timing circuit (called a *watchdog timer*) that is activated during each bus cycle. If the cycle does not terminate after a reasonable number of clock cycles, the timer asserts READY# and causes an interrupt.

Note that ADS# can be deactivated at the end of T1, whereas the bus control signals remain valid until close to the end of the entire cycle. In write cycles, output data becomes valid on the data bus at the start of phase 2 in T1.

Pipelined Bus Cycles

Pipelined bus cycles allow overlapped signals as shown in Figure 8-9. The second half of each cycle (designated T2P in Figure 8-9) is the time for both data transfers and establishing address and control signals for the next cycle.

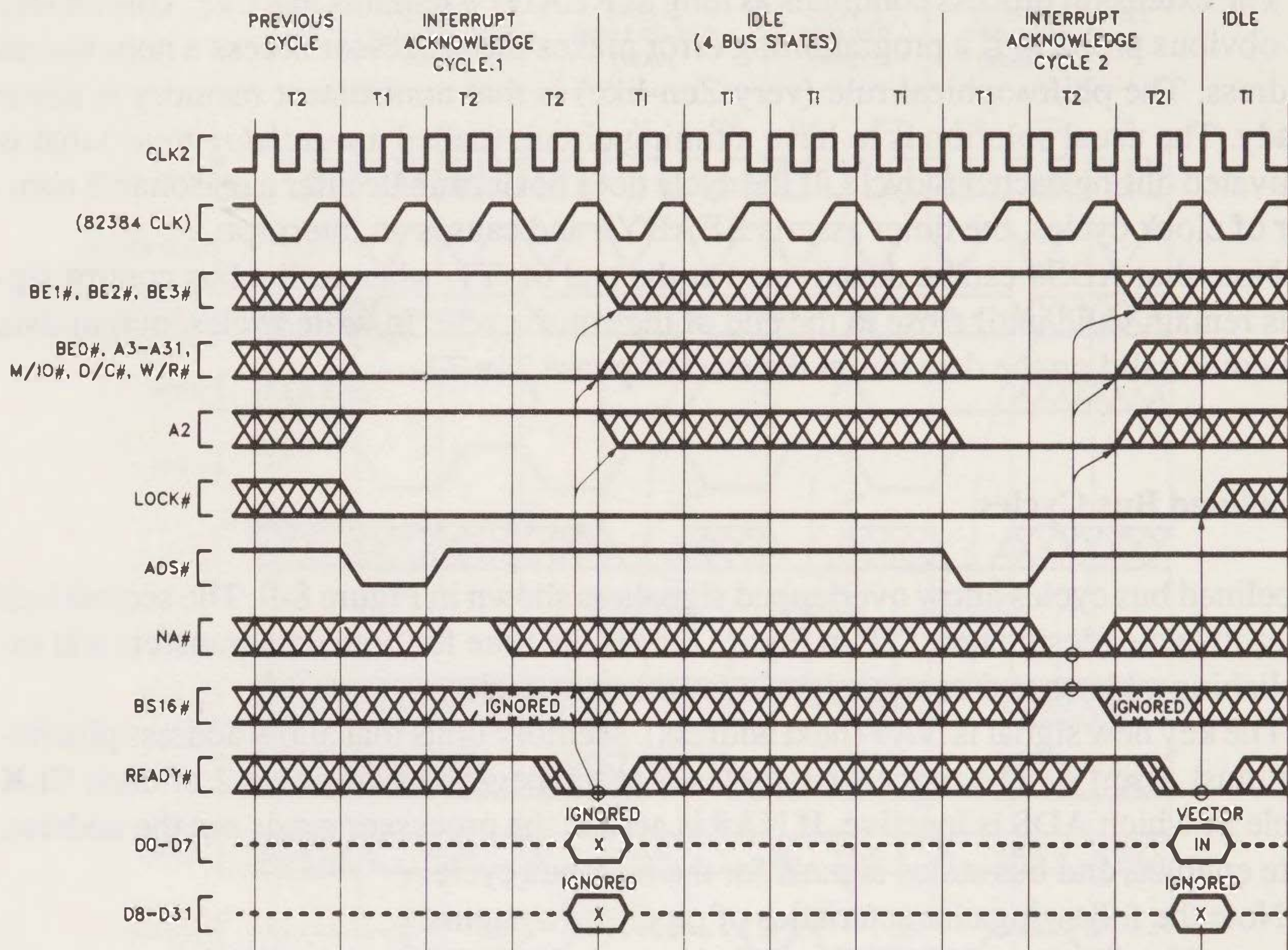
The key new signal is NA# (next address). Memory units that allow address pipelining must assert it. The processor samples it at the beginning of phase 2 of each CLK cycle in which ADS is inactive. If NA# is active, the processor sends out the address, byte enables, and bus status signals for the next bus cycle.

Note the following characteristics of pipelined systems:

- The first bus cycle after an idle bus state is always nonpipelined.
- The bus cycle in which NA# is first recognized must be extended by at least one CLK cycle to allow the output of address and status before its end. There is thus some initial overhead that occurs after any idle state.

Interrupt Acknowledge Cycles

The 80386 does special bus cycles in response to an INTR interrupt. They serve as an acknowledgment to external devices such as an 8259 interrupt controller. At this time, the controller can put a vector (the *interrupt type*) on the data bus for the CPU to read. Figure 8-10 contains timing diagrams for the interrupt acknowledge cycles. There are two of them, separated by a gap. M/IO#, D/C#, and W/R# are all low during the cycles. External circuitry such as programmable array logic usually decodes this state to form an interrupt acknowledge output.



Interrupt Vector (0-255) is read on D0-D7 at end of second Interrupt Acknowledge bus cycle.

Because each Interrupt Acknowledge bus cycle is followed by idle bus states, asserting NA# has no practical effect. Choose the approach which is simplest for your system hardware design.

Figure 8-10

Interrupt acknowledge bus cycles.

Note that the interrupt control circuitry must ensure a proper response to the acknowledge cycles. The 80386 does not supply any way to distinguish them. Instead, it always provides the same address:

BE0# low

BE1#, BE2#, and BE3# high

Table 8-7

80386 Performance with Wait States and Pipelining

Wait States When Address is Pipelined	Wait States When Address is Not Pipelined	Performance Relative to Non-Pipelined 0 Wait-State	Bus Utilization
0	0	1.00	73%
0	1	0.91	79%
1	1	0.81	86%
1	2	0.76	89%
2	2	0.66	91%
2	3	0.63	92%
3	3	0.57	93%

Address 4 ($A_2 = 1$, everything else is 0) during the first cycle, address 0 during the second cycle (see Figure 8-10).

The vector transfer always occurs on D0 through D7 at the end of cycle 2, as shown in Figure 8-10. The processor automatically puts four idle bus states between the two cycles to synchronize with an 8259 priority interrupt controller. $ADS\#$ is active at the start of each interrupt acknowledge cycle, and the control circuitry must respond by asserting $READY\#$.

BUS PERFORMANCE CONSIDERATIONS

The actual performance of 80386 bus cycles depends on the number of wait states required and on whether the address is pipelined. Table 8-7 shows simulated results for different numbers of wait states and different pipelining conditions. Address pipelining reduces the need for wait states. Because of the overlap, an access requiring two wait states without pipelining will require only one wait state with it. This can increase throughput significantly without requiring faster memory.

COPROCESSORS

The 80386 can use either an 80287 or an 80387 numeric coprocessor. Both are software compatible with the popular 8087 coprocessor, often used with 8086 or 8088 CPUs. The 80386 can also use the high-speed Weitek WTL1167 floating point chip set. These devices perform numeric instructions in parallel with the 80386, specializing particularly in 80-bit floating point arithmetic defined by IEEE Standard 754. The 80287 is a 16-bit device, whereas the 80387 is a 32-bit device. The 80387 also runs at higher clock speeds than the 80287 and has extra trigonometric functions (sines and cosines).

The key instruction for a coprocessor is ESC, designated by the binary pattern 11011 in the five most significant bits. The 80386 sends ESC instructions on to the coprocessor through I/O addresses 800000F8H and 800000FCH. The execution does not depend on the 80386's I/O privilege level.

The coprocessor addresses are completely separated from conventional I/O addresses by having A31 high. Note that the 80386 sends address lines A16 through A31 low when accessing ports in its standard 64K I/O space. Of course, the I/O section must decode A31 to avoid conflict between coprocessor cycles and I/O cycles.

The 80386 cannot transfer an instruction to the coprocessor until the BUSY# input is inactive (high). The 80386 knows when to transfer data to or from the coprocessor because PEREQ (coprocessor request) goes high.

80287 Numeric Coprocessor Interface

Figure 8-11 shows a typical interface between an 80386 microprocessor and an 80287 numeric coprocessor. The 80287 is selected when M/IO# is low and A31 is high. That is, it is selected during I/O cycles only if A31 = 1. This works if all other I/O ports are specifically selected by A31 = 0. Note that data transfers occur only over the lower half of the data bus, as the 80287 is a 16-bit device. The 80287 requires its own clock generator, as it runs much slower than an 80386 and does not use a double-frequency clock input. The difference in clock rates and timing also makes some latches necessary.

In response to an ESC instruction, the 80386 does one or more I/O cycles to the 80287's ports. The 80386 automatically converts 32-bit memory transfers into 16-bit transfers, and vice versa. That is, it converts 32-bit transfers to the 80287 into two successive 16-bit transfers and combines 16-bit transfers from the 80287 into 32-bit memory transfers. This happens automatically — there is no need to activate the 80386's BS16# input.

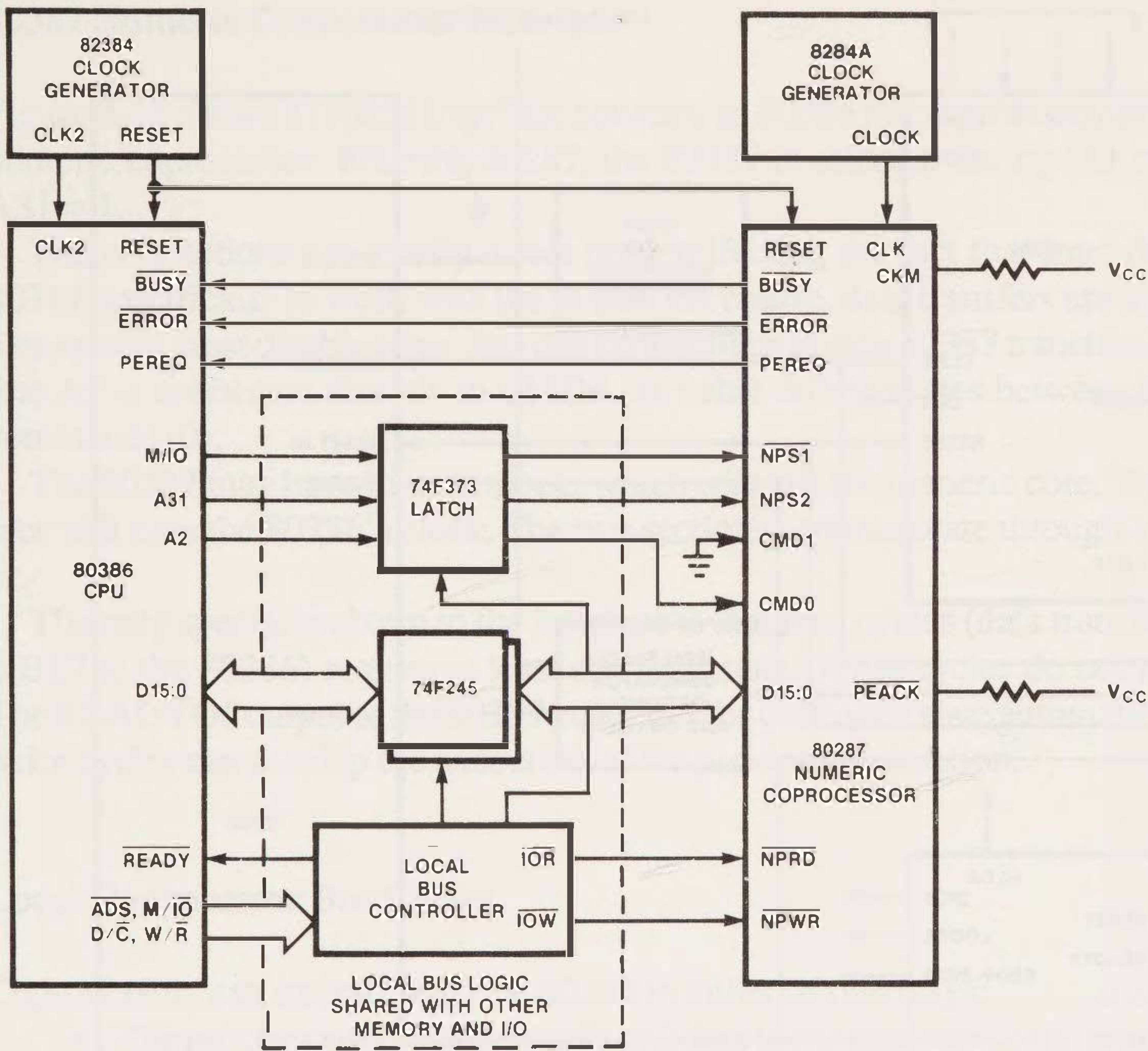


Figure 8-11
80386 system with 80287 coprocessor.

The 80287 uses its command inputs (CMD0 and CMD1) to differentiate between data and commands. The interface in Figure 8-11 has the CMD lines connected to ground (CMD1) and to A2 (CMD0), respectively. The 80287 thus interprets outputs to address 800000F8H (A2 = 0) as commands, and those sent to address 800000FCH (A2 = 1) as data.

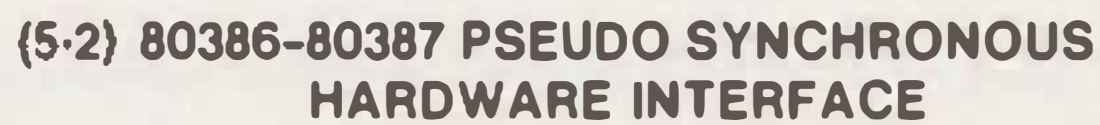


Figure 8-12
80386 system with 80387 coprocessor.

80387 Numeric Coprocessor Interface

Figure 8-12 shows a typical interface between an 80386 microprocessor and an 80387 numeric coprocessor. Like the 80287, the 80387 is selected during I/O cycles when $A_{31} = 1$.

The connections are mostly direct here, reflecting the fact that Intel designed the 80387 specifically to work with the 80386. Of course, data transfers are a full 32 bits. The system must disable other data bus connections during 80387 transfers. As address line A_2 is connected directly to $CMD\#$, its value differentiates between data (1) and commands (0).

The 80387 may have its own clock, which controls the numeric core. The bus interface unit uses the 80386's clock. The two sections communicate through a FIFO buffer.

The only special problem in the interface is that read cycles (data transfers from the 80387 to the 80386) require at least one wait state. Write cycles do not require this. The $READYO\#$ output of the 80387 generates the extra wait state automatically. 80387 write cycles can overlap the execution of the previous instruction.

Local Coprocessor Bus Cycles

Coprocessors can interact with the 80386 in either of two ways:

- The processor sends commands and data to the coprocessor as part of the execution of an ESC instruction.
- The coprocessor requests data transfers using the PEREQ signal.

Note that, in the ESC case, the 80386 sets an internal memory address base register, memory address limit register, and direction flag. The coprocessor can then request operand transfers by activating PEREQ. This can happen only while the coprocessor is executing an instruction.

Operand transfers may take a long time. For example, the operands may be misaligned, thus requiring extra cycles. Furthermore, operands may be too long for the 80287's 16-bit bus or even for the 80387's 32-bit bus. In particular, IEEE 754 double-precision floating point numbers are 64 bits long (sign, 11-bit exponent, and 52-bit significand).


```

; initialization routine to detect an 80287 Numeric Processor

FND_287: FNINIT          ; initialize Numeric Processor
        FSTSW AX         ; retrieve 80287 status word
        OR AL, AL        ; test low-byte 80287 exception flags
                        ; if all zero, then 80287 present and
                        ; properly initialized
                        ; if not all zero, then 80287 absent.
        JZ GOT_287       ; branch if 80287 present

        SMSW AX          ; No numeric processor,
        OR AX, 04H       ; set EM bit in machine status word
        LMSW AX          ; to enable software emulation of 80287
        JMP CONTINUE

GOT_287: SMSW AX          ; Numeric Processor present
        OR AX, 02H       ; set MP bit in machine status word
        LMSW AX          ; to permit normal 80287 operation

CONTINUE:                ; and off we go. . .

```

Figure 8-13

Routine to detect the presence of an 80287 numeric coprocessor.

80287/80387 Recognition

The basic way for an 80386 processor to determine whether an 80387 coprocessor is present is to test the initial state of the ERROR# input. The 80387 makes ERROR# active after RESET. The processor checks it automatically at that time (and before executing the first instruction). If ERROR# is active, the processor sets the ET bit (bit 4 of control register 0). All the program must do is execute an FINIT instruction to reset the 80387's ERROR# output.

If ERROR# is inactive, the processor clears the ET bit. Note that ET = 1 means that an 80387 is present. ET = 0 means only that an 80387 is not present. There could be an 80287 in the circuit or no coprocessor at all. A routine like the one shown in Figure 8-13 can tell these alternatives apart by reading the 80287's status word. The result will be FFFF hex (all ones) if no 80287 is present, as the lines will be floating.

If there is no 80287 available, the processor must set the Emulate Coprocessor (EM) bit (bit 2 of Control Register 0). The processor will then emulate coprocessor instructions in software; this maintains compatibility between systems with coprocessors and those without them. Of course, systems that must emulate the coprocessor instructions will run much more slowly. If an 80287 is present, the processor must set the MP (Math Present) bit. MP is bit 1 of Control Register 0 (see Figure 2-4).

Coprocessor Exceptions

There are three coprocessor exceptions:

- Interrupt 7 — Coprocessor not available
- Interrupt 9 — Coprocessor segment overrun
- Interrupt 16 — Coprocessor error

Interrupts 7 and 16 are benign exceptions (see Table 7-4) that software can handle. Interrupt 9 is a contributory exception which should be avoided. Interrupt 13 (a contributory exception) can also occur if an operand lies outside the segment limit or violates some other protection restriction.

Two situations can cause interrupt 7:

- The processor tries to execute an ESC instruction when the EM bit of control register 0 is set. The exception handler must direct the processor to the software emulation of the instruction.
- The processor tries to execute either a WAIT or an ESC instruction with both the MP (Math Present) and TS (task switched) bits set. This means that the processor has switched tasks since the last time it used the coprocessor. The exception handler must determine if the current task is the same as the one that was executing the last time the coprocessor was used (that is, the processor has returned to it after switching away). If not, the handler must save the coprocessor's context in the previous task's TSS.

Interrupt 9 occurs in protected mode if an operand of a coprocessor instruction wraps around an addressing limit or spans inaccessible addresses. Proper alignment of operands will eliminate this problem.

Interrupt 16 occurs if the coprocessor detects an exception condition during instruction execution. The 80386 recognizes the problem when it checks ERROR# at the beginning of a WAIT or certain ESC instructions. The handler must examine the coprocessor's status register to determine the cause of the condition.

MEMORY INTERFACING

The basic interface for 80386 memory sections consists of the following:

- Control signal generation circuitry to provide memory, I/O, interrupt, and other control signals.

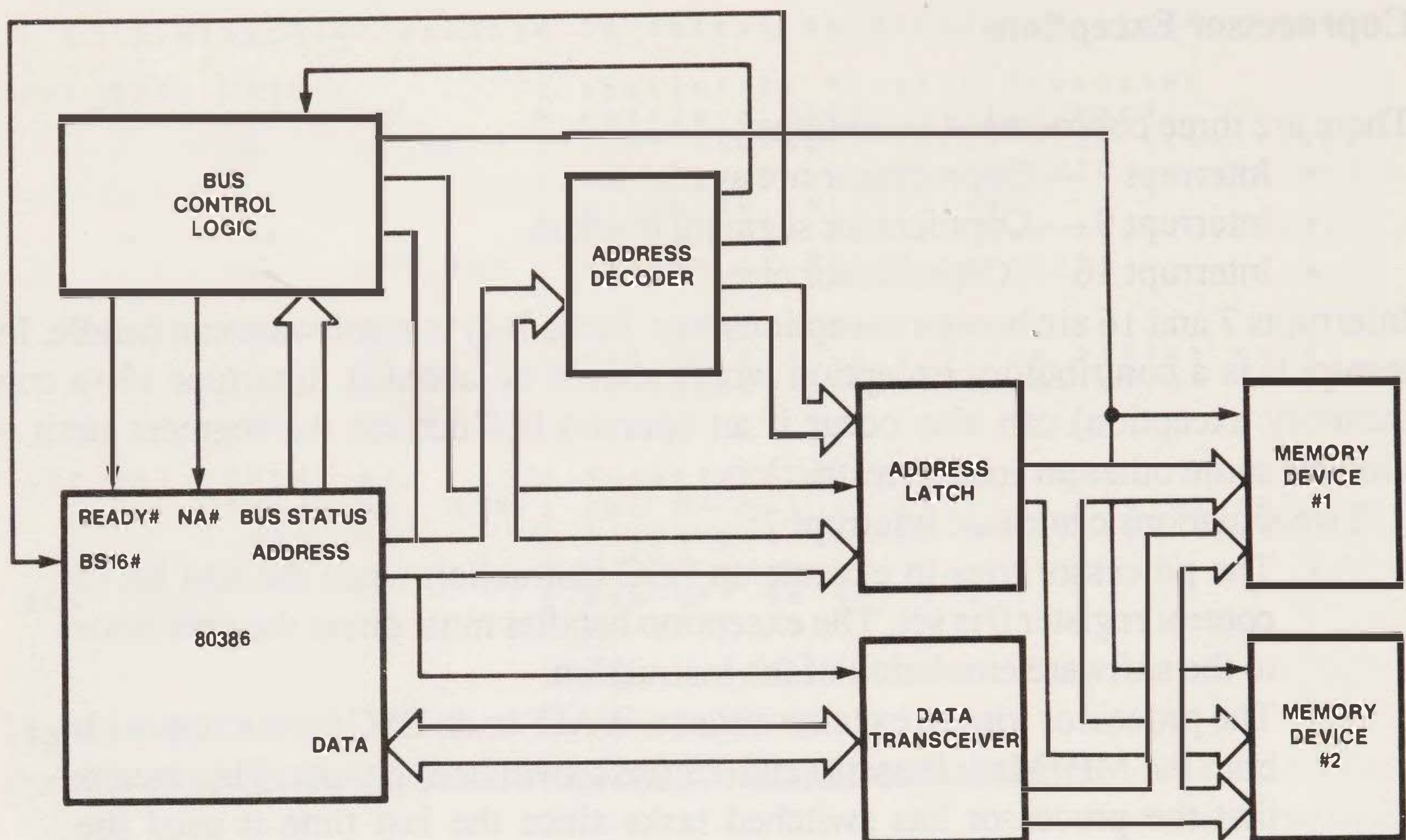


Figure 8-14

Basic memory interface block diagram.

- Address latches to hold addresses beyond the time they are usually on the bus. Latches are essential in systems that use address pipelining.
- Bus buffers to provide more drive current and better signal isolation.
- Data bus control circuitry to prevent bus contention. Bus contention means that more than one device is trying to control the bus at a given time. It is usually the result of a device being relatively slow to get off the bus.
- Address decoding circuitry to select memory banks or boards and I/O ports.

Figure 8-14 shows the basic memory interface. The circuitry may consist of individual TTL devices, programmable array logic (PALs), or PROMs. The bus control logic is usually a series of PALs as shown in Figure 8-15. The interface is asynchronous; that is, it depends on the exchange of status and control signals rather than on a clock.

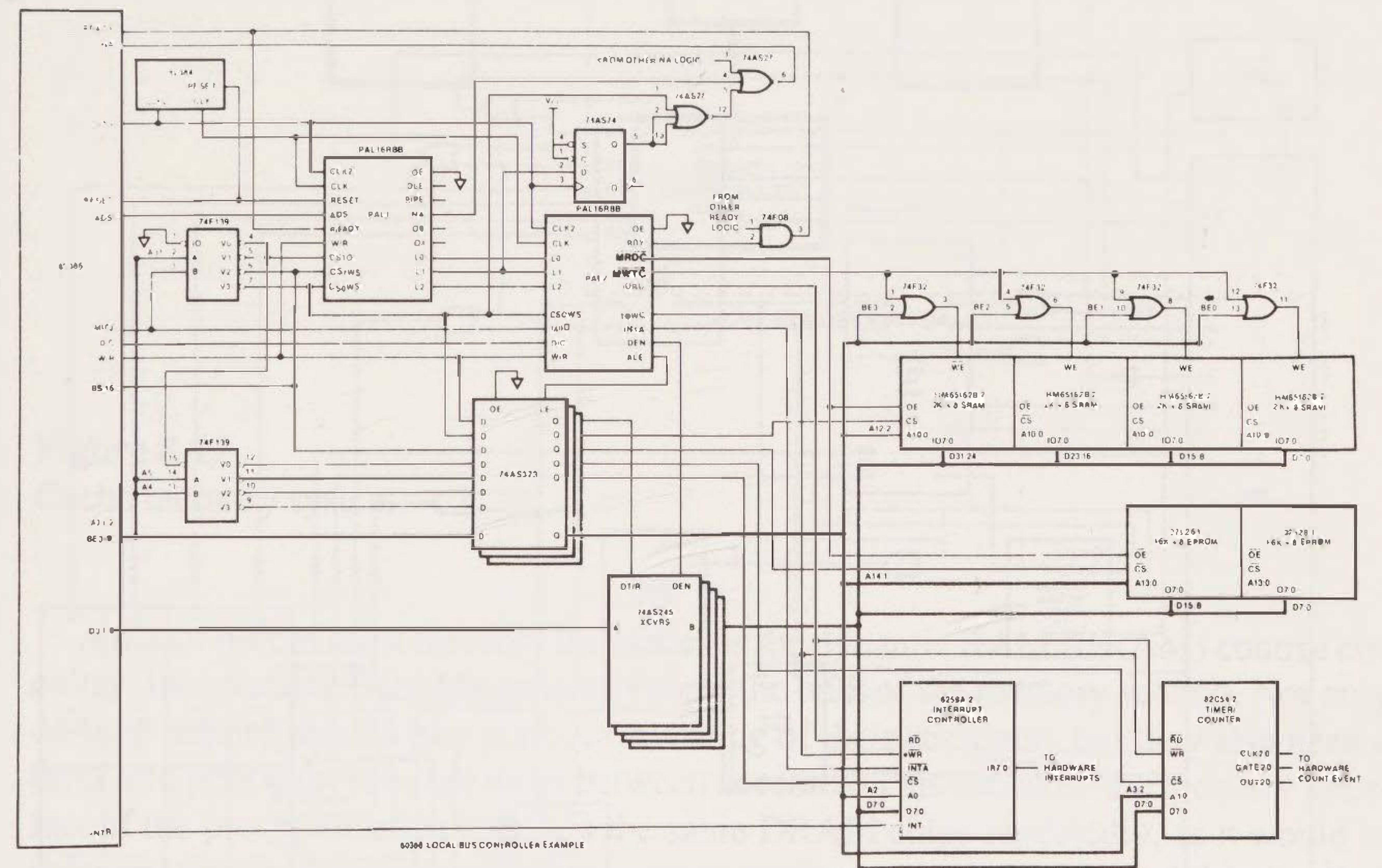


Figure 8-15
Bus control logic.

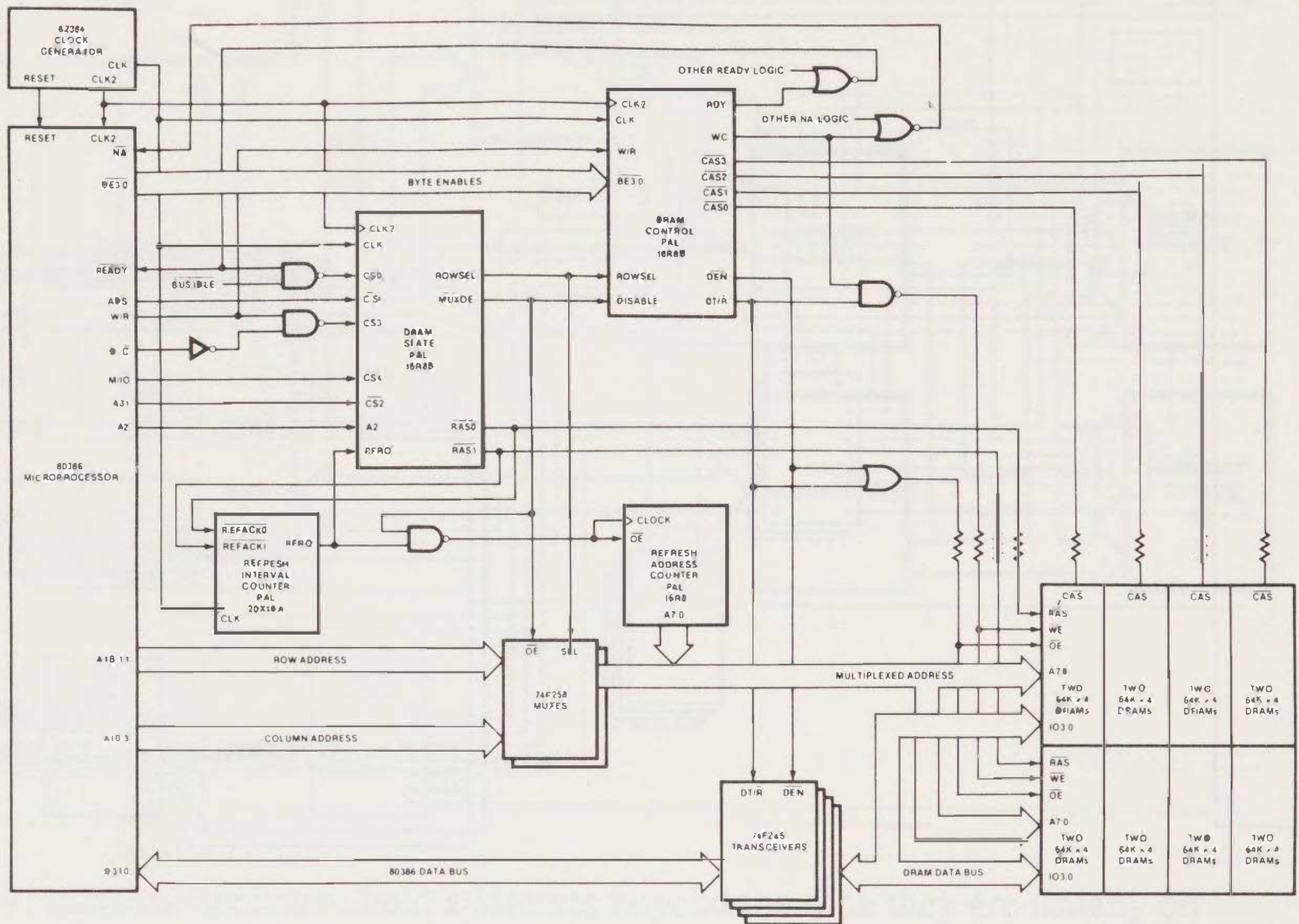


Figure 8-16
Controller for a 3-CLK DRAM controller.

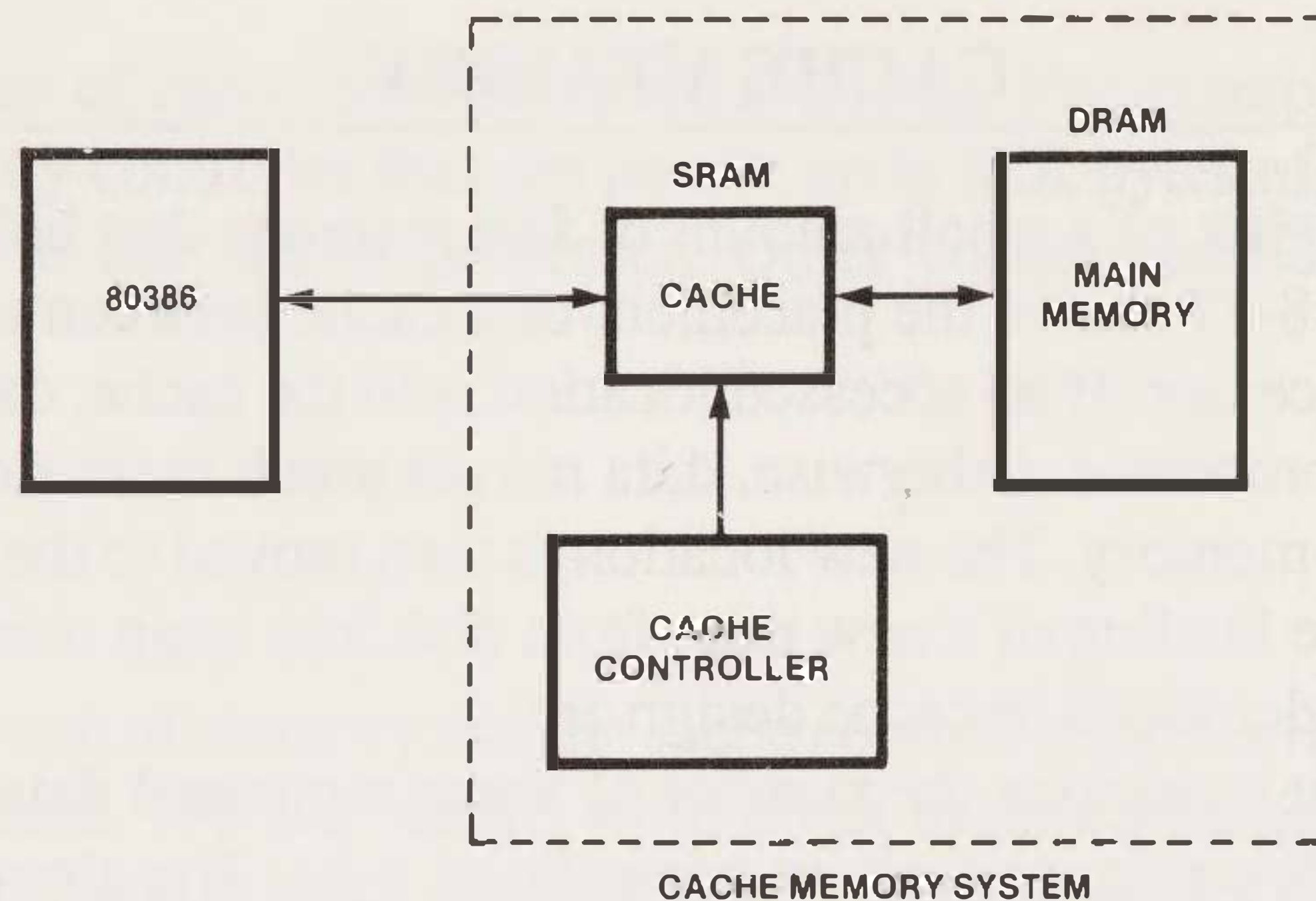


Figure 8-17
Cache memory system.

A major part of most memory interfaces is the dynamic RAM (DRAM) control circuitry. Inexpensive DRAMs generally form the bulk of the memory section. Not only do they require *refresh* (the periodic rewriting of their contents), but they also need a brief idle period (*precharge time*) between accesses. This can slow the memory interface if the processor often accesses the same DRAM chips repeatedly, as it would in the usual arrangement when reading instructions from consecutive or slightly separated addresses in memory.

One way to avoid the need for idle time is to direct successive memory accesses to different banks. That is, the next consecutively addressed double word is always in a different bank of memory with its own control circuitry. While such an arrangement seems odd, it makes no difference to the processor, any more than having successive street addresses on opposite sides of the street affects mail delivery. We refer to the alternating arrangement as *interleaved memory*. Figure 8-16 shows a typical circuit in which a PAL selects DRAMs, manages refresh, and keeps track of which banks require precharge time.

Interleaved memory complicates system debugging. A failure in one memory bank causes a problem only with alternating double word addresses. This is clearly much more difficult to diagnose than is a problem that affects consecutive addresses.

CACHE MEMORY

Cache memory consists of a small amount of fast memory that holds frequently accessed data. Figure 8-17 shows the placement of a cache between main memory and the 80386 microprocessor. If an accessed location is in the cache, data moves quickly between it and the processor. Otherwise, data moves much more slowly between the processor and main memory. The new location is then moved to the cache, much as a page fault causes the loading of a new page from disk into main memory.

Among the considerations in cache design are:

- How do you maximize the number of times requested data is found in the cache? We refer to such an occasion as a *hit*. The alternative is, of course, a *miss*. We call the percentage of hits the *hit ratio*.
- How do you quickly determine whether a particular location is in the cache?
- What is the optimum size of the units or blocks of memory in the cache? Obviously, most programs access sets of contiguous locations, and the processor might as well move them all to the cache at once.
- How do you handle the changing (writing) of cache locations? New data written into the cache must eventually be written into main memory as well. Furthermore, external bus masters and DMA controllers must update the cache as well as main memory if they make changes. The problem is like deciding when to save a document from RAM to disk.

Of course, caches do not produce something for nothing. There is always some reduction of performance, particularly during write cycles. Fortunately, writes are far less common than reads. Caches also require extra hardware for control and decoding.

Cache Controller

A cache controller must manage the cache memory system. This device does the following tasks:

- It determines whether a particular address is in the cache. It does this by using identification markers (*tags*) attached to each block.
- It accesses the cache if the address is there (that is, if a hit occurs).
- It moves a block of data from main memory to the cache if the address is not there (that is, if a miss occurs).
- It keeps track of changes to the cache and to main memory that is cached.

LSI implementations of cache controllers are available. For example, Intel offers the 82385 cache-memory controller that can handle up to 32K bytes of cache memory. It interfaces directly to the 80386 processor. Other cache controllers have on-chip memory.

Block Size

A block is the basic unit of memory that the cache controller moves from main memory into the cache at a time. When a needed word is not in the cache, the controller loads not only it but also the entire block containing it. Typical sizes for blocks are 4, 8, and 16 bytes.

Obviously, a larger block increases the hit rate if the processor is accessing consecutive addresses or repeating a short loop. On the other hand, larger blocks take longer to move, reduce the number of blocks that fit in a cache, and increase the likelihood of unneeded data being placed in the cache. After all, the processor does not access consecutive addresses forever. There are branches, jumps, and widely separated accesses to consider.

Cache Organization

Common cache organizations are:

- Fully associative, in which each data block has a tag that completely identifies its contents. There are no restrictions on where addresses can go. Any memory block could end up in any cache block. To determine whether an address is in the cache, the controller simply compares it (or part of it) with each tag.
- Direct mapped, in which each cache block can only hold one of a restricted set of memory blocks. The tag then identifies which memory block it actually holds. To determine whether an address is in the cache, the processor must only check whether its tag is in the position corresponding to the block that could hold it. The address cannot be cached anywhere else, so no further comparisons are necessary.

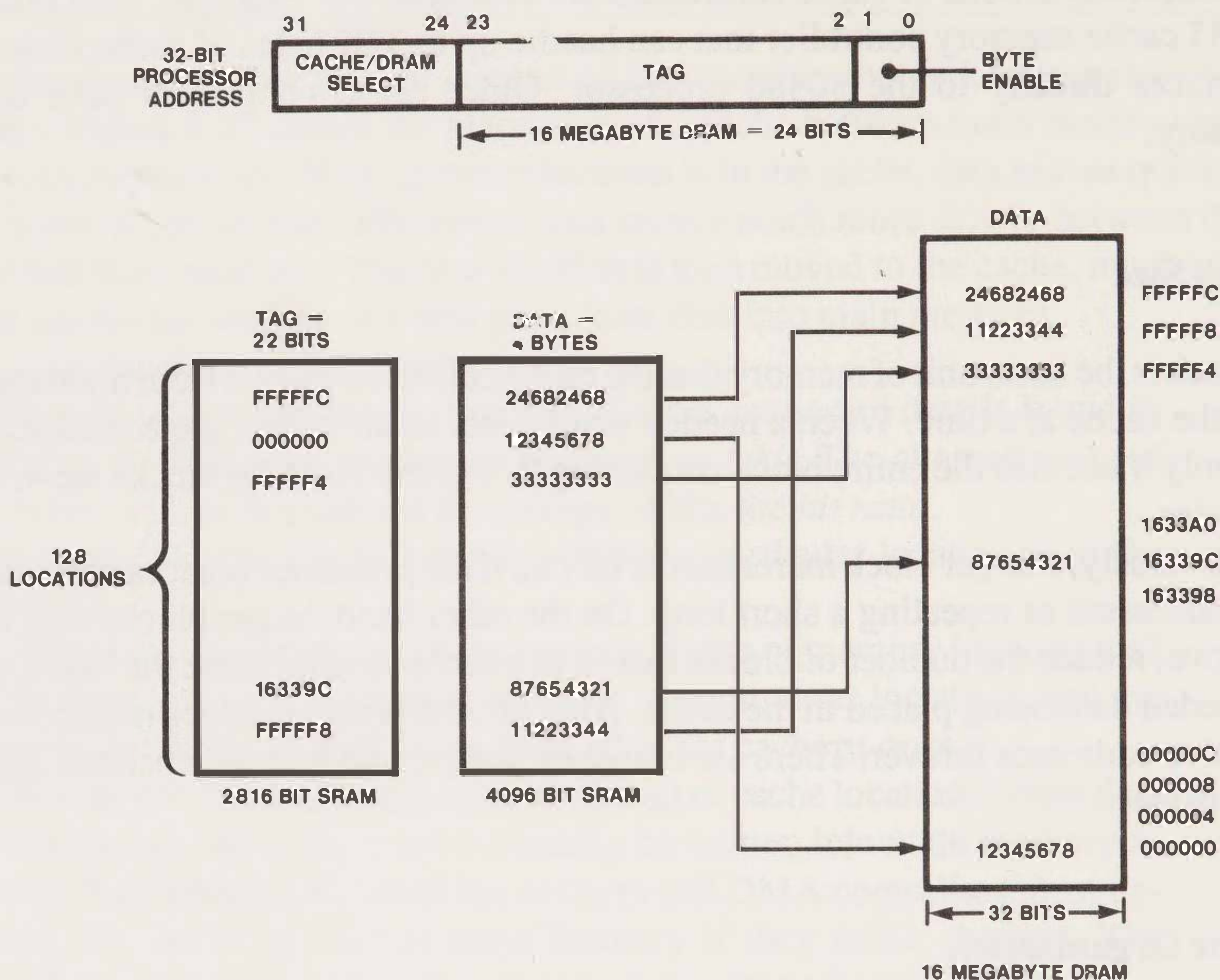


Figure 8-18
Fully associative cache organization.

- Set associative, in which the cache consists of several direct mapped caches, one of which is selected on an associative basis. This organization is clearly a hybrid between the direct mapped cache and the fully associative cache.

The fully associative cache provides the highest hit ratios and the least traffic between main and cache memory. After all, there are no restrictions on what addresses the cache can contain simultaneously. So there are no patterns of accesses that can cause special problems. Figure 8-18 shows a fully associative cache system with 512 bytes of cache memory (a very small amount) and 16 Mb of main memory. The proces-

80386 Hardware Features

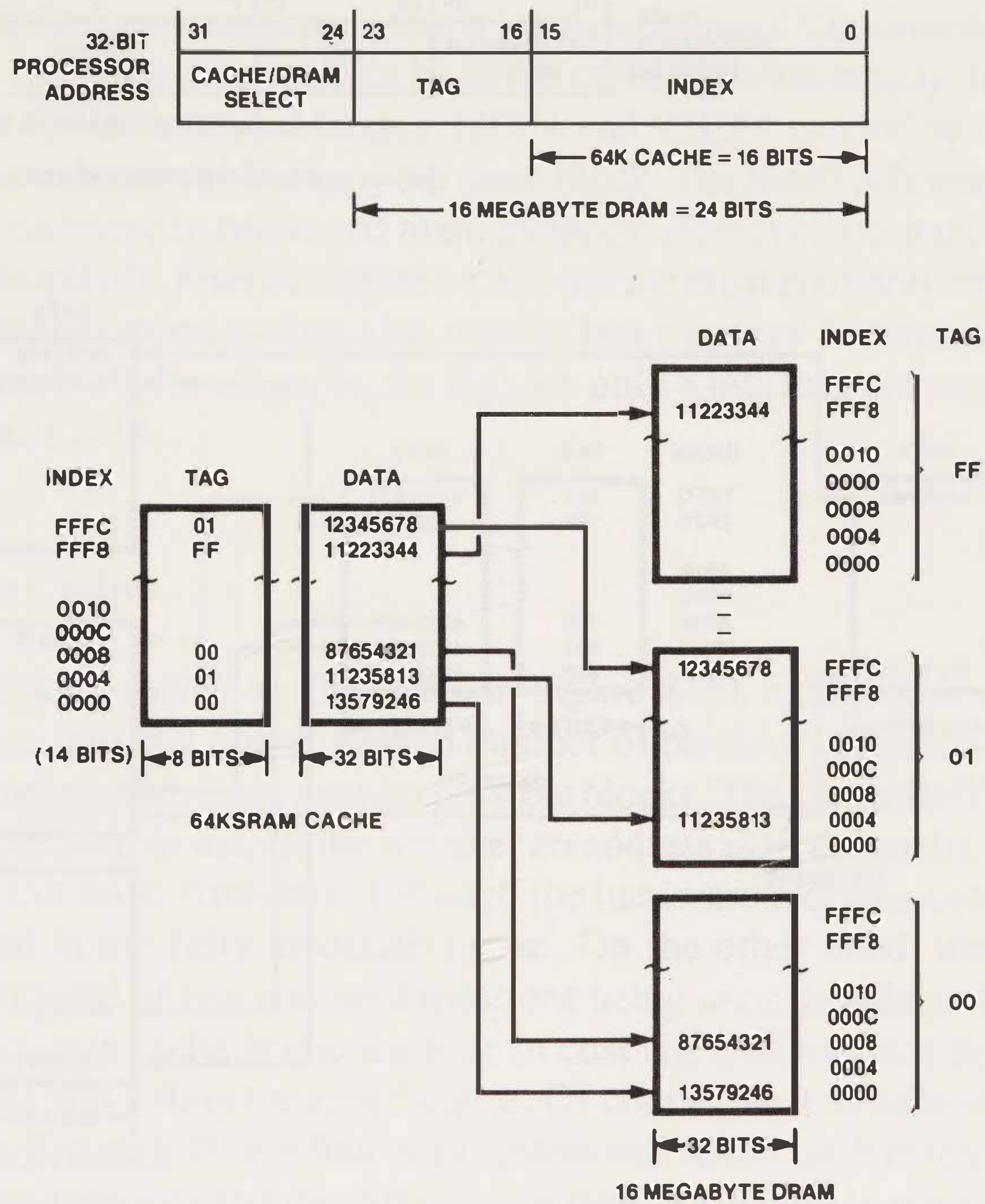


Figure 8-19
Direct mapped cache organization.

processor may have to perform up to 128 22-bit comparisons to determine whether a location is in the cache. Clearly, this will be either quite time consuming (if slow memory is used) or expensive (if fast memory is used). Thus fully associative caches are usually impractical. New hardware such as content-addressable memories may change this situation in the future. Note that a more realistically sized cache (say, 16 Kb) would require even more comparisons (4096). It would also require 11 Kb for tags.

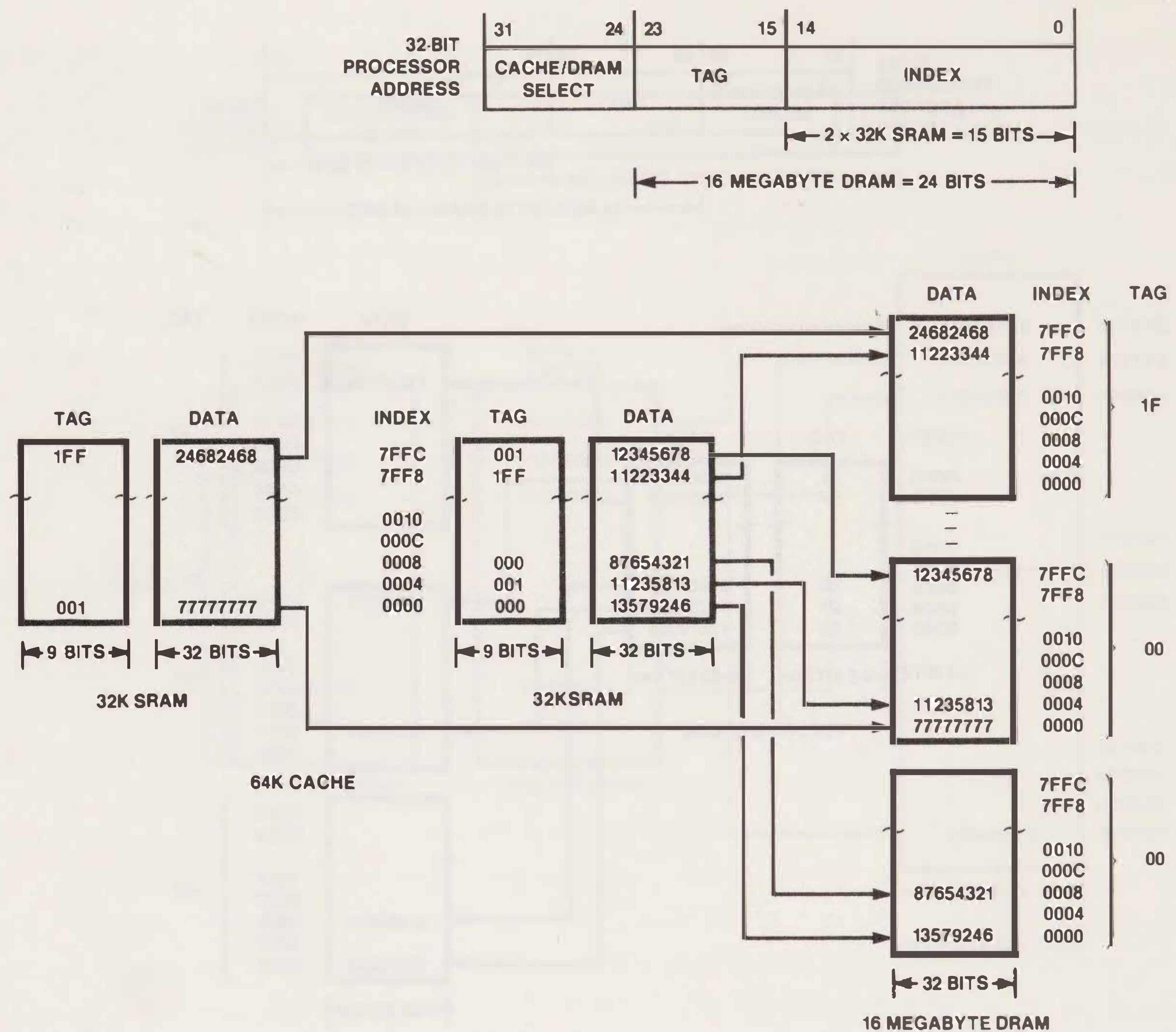


Figure 8-20
Two-way set associative cache organization.

Direct Mapped Cache

In a direct mapped cache such as the one shown in Figure 8-19, only one comparison is necessary. An address with a particular index (say, 3EF4 in the example shown) can only be at a particular address (3EF4) in the cache. All the cache controller must do is compare the cache location's tag with the original address' tag. If they are the same, the address is in the cache. Here the tag is just bits 16 through 23 of the memory address.

The drawback to direct mapping is that it restricts the possible contents of the cache. Locations with the same index cannot be in the cache simultaneously. In the example in Figure 8-19, for instance, addresses 13EF4 and 23EF4 cannot be cached simultaneously because they would occupy the same block. The result is fewer hits and more movement of data between cache and main memory. However, because of their speed and hardware simplicity, direct-mapped caches are the most common implementation.

Note that direct-mapped caches also require less memory for tags, as the tags are shorter. In Figure 8-19, for example, the tags are only 8 bits long, whereas they are 22 bits long in Figure 8-18.

Set Associative Caches

In the set associative cache, as illustrated in Figure 8-20, a particular main memory location could go into any one of a small number of blocks in the cache. Figure 8-20 shows a situation in which there are two possible blocks. The controller must therefore make two comparisons to determine whether an address is in the cache. This requires either extra time or more hardware, although the hardware requirements are far more manageable than in the fully associative case. On the other hand, the associativity reduces the likelihood of two accessed locations being uncacheable at the same time. Thus a set associative cache is intermediate in cost and performance between a fully associative cache and a direct-mapped cache. Of course, we can extend the two-way system shown in Figure 8-20 to a four-way system and so on. Each extension increases the hardware requirements and the difficulty of deciding where to put new blocks.

Cache controllers can implement the associativity hardware in LSI form. For example, Intel's 82385 device allows two-way set associative organizations. Other controllers such as the NEC μ PD43608R and the Austek Microsystems A38152 allow four-way systems.

Cache Updating

The updating system ensures that data in the main memory and data in the cache agree eventually. This avoids the problem of "stale" data as shown in Figure 8-21. Stale data is worse than stale bread; its shelf life is shorter and the odor is unbearable. Two common approaches to updating are:

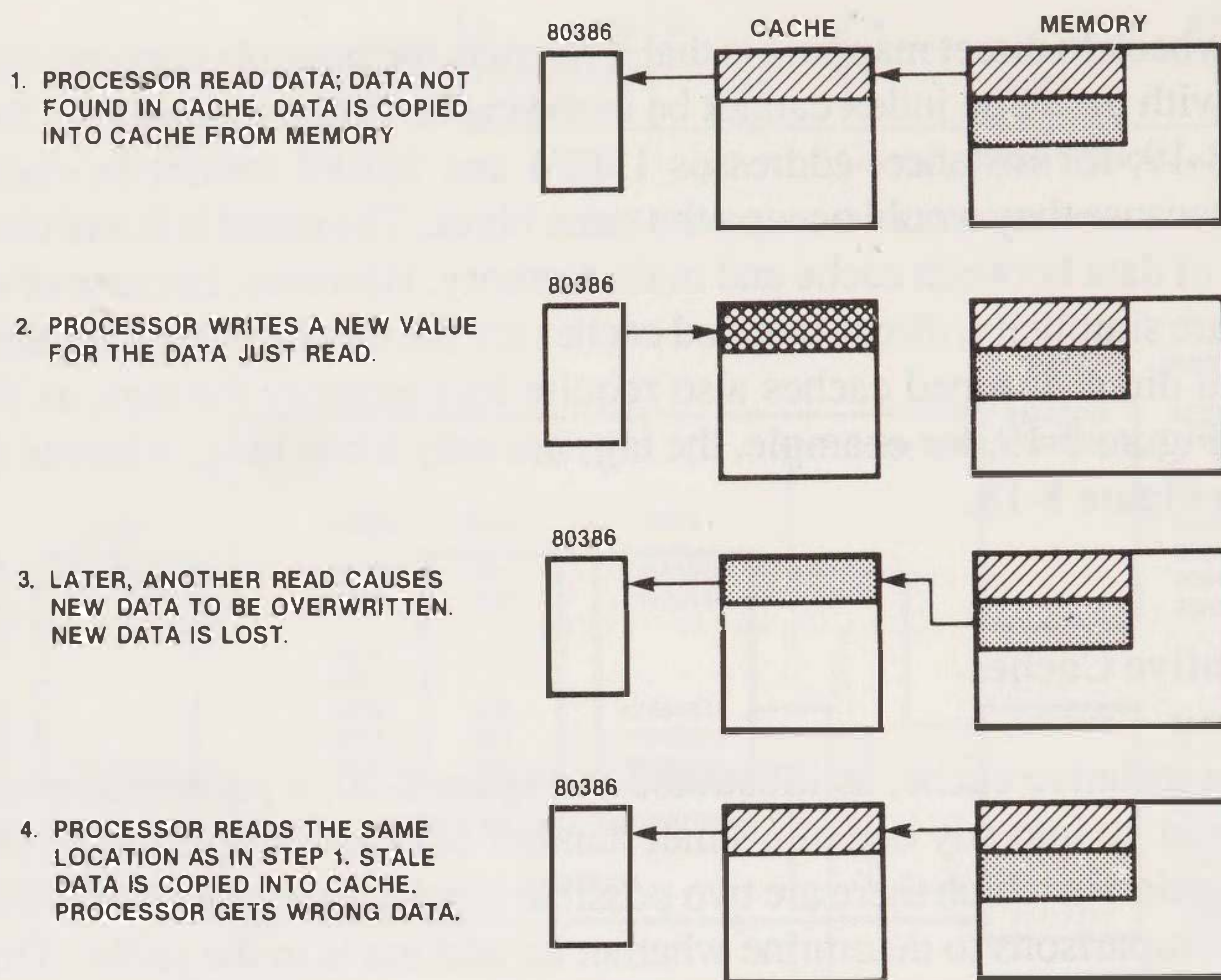
**Figure 8-21**

Illustration of the stale data problem.

- Write through in which the cache controller immediately writes the data into main memory.
- Write-back in which the cache controller writes data into main memory only when removing a location that has been changed. There is no reason to change main memory each time the cache changes.

In a write-through system, the controller writes data to the main memory immediately after writing it into the cache. This approach is simple and ensures that the contents of the main memory are always valid. On the other hand, the accesses of main memory take extra time and occupy the buses. One way to reduce time consumption is by buffering write-throughs. The processor can then begin a new cycle before the writing of the main memory is completed. The only problem comes when two consecutive cycles require main memory accesses either to write data or to handle a cache miss. Then the

processor must wait for the previous main memory cycle to end before starting the next one. Obviously, the buffering also increases circuit complexity and adds to the actual cycle time (because of the buffer delays).

In a write-back system, the tag field of each cache block contains an altered or “dirty” bit. This bit is set if new data has been written into the block. It therefore indicates that the cache contains different data from the main memory.

The procedure in a write-back system is to check the altered bit first. If it is set, the controller writes the block to main memory before loading new data into the cache. The advantage of write-back is that it reduces the number of times main memory has to be written. Only locations that are changed and then removed from the cache need to be written back into main memory. On the other hand, main memory is not necessarily up to date. It may, in fact, contain data from sometime ago. Write-back also requires a more complex controller than does write-through. Of course, controller complexity is largely irrelevant for LSI implementations.

Noncacheable Memory

Some memory locations cannot be cached. These may include:

- Memory-mapped I/O devices. The cache locations would not reflect external changes. For example, imagine if a memory-mapped keyboard were cached. The processor would never see subsequent keystrokes, as it would read only the cached location.
- Interrupt vectors. These are generally accessed so seldom that they are not worth caching anyway.
- Memory shared by several processors.

One way to differentiate between cacheable and noncacheable memory is by decoding some of the more significant address lines. For example, in Figures 8-18 through 8-20, address bits 24 through 31 are unassigned. These bits could be decoded to select either the 16 Mb of cached memory or other noncacheable memory. A simple approach is to just use A31 for selection. It could choose between 2 Gb of cacheable memory and 2 Gb of noncacheable memory.

Noncacheable memory is one way of allowing several processors to share memory. Cacheable shared memory can result in stale data, a problem we refer to as maintaining *cache coherency*. On the other hand, frequent accesses to noncacheable memory can slow the system significantly. A partial solution to the problem is to copy data to cacheable memory as long as only one processor is accessing it. Maintaining coheren-

cy is also a challenge for systems with DMA controllers. The Intel 82385 cache controller helps by detecting writes to cached memory and invalidating the locations. Intel refers to this activity as (so help me!) “snooping.”

Cache Performance

Table 8-8 contains performance data for various sizes of direct-mapped and set associative caches. As mentioned earlier, fully associative caches are currently impractical in most situations. Clearly the hit ratios and performance ratio increase with cache size. However, the improvement is marginal (about 3 percent in hit rate and about 0.7 percent in performance ratio) if the cache is 32K or above. The differences between direct and set associative caches are small for all sizes shown (32K, 64K, and 128K). The payoff would not justify the added hardware complexity. A more profitable approach is to increase the line or block size from 4 to 8 bytes. Comparing the performance ratios to the bottom two lines of the table shows that a 64K direct-mapped cache gives about 95 percent of the performance of a system with all high-speed memory (static or SRAM).

SUMMARY

The 80386 has an asynchronous memory and I/O interface. That is, it exchanges status and control signals (called *handshaking*) rather than depending on a clock. The processor asserts Address Status (ADS#) when a valid address is on the bus. The memory or I/O section must respond by asserting READY# when either data is available for reading or a write has been completed. The READ/WRITE#, DATA/CONTROL#, and MEMORY/IO# signals can be used for bus control, cycle identification, and decoding. The NEXT ADDRESS# signal indicates that the memory can accept another address; it allows overlapping of memory cycles, thus reducing the need for wait states.

The 80386 provides a RESET input, two interrupt inputs (maskable and nonmaskable), DMA request (HOLD) and acknowledge (HLDA), and a multiprocessor LOCK signal. It acknowledges interrupts by performing special bus cycles to obtain an interrupt vector.

The 80386 can use either an 80287 or an 80387 numeric coprocessor. The 80287 is a 16-bit device, whereas the 80387 is a 32-bit device. The processor sends commands

to the coprocessor through special I/O port addresses. The coprocessor requests data transfers by activating the processor's PEREQ input.

Among the ways to reduce memory cost in 80386-based systems while still retaining high performance are:

- Interleave memory addresses so that successive accesses refer to different banks. This avoids setup (precharge) time.
- Keep frequently used data in a small amount of high-speed cache memory. A cache controller must determine whether a particular address is in the cache, access the cache, move data from main memory to the cache, and keep track of changes to the cache and main memory.

The use of cache memory introduces several new problems, such as deciding how and when to write new data into memory and how to maintain coherency between cache and main memory.

Appendix

A

80386 Instruction Set

INSTRUCTION SET

This section describes the 80386 instruction set. A table lists all instructions along with instruction encoding diagrams and clock counts. Further details of the instruction encoding are then provided in the following sections, which completely describe the encoding structure and the definition of all fields occurring within 80386 instructions.

80386 INSTRUCTION ENCODING AND CLOCK COUNT SUMMARY

To calculate elapsed time for an instruction, multiply the instruction clock count, as listed in Table 8-1 below, by the processor clock period (e.g. 62.5 ns for an 80386-16 operating at 16 MHz (32 MHz CLK2 signal)).

For more detailed information on the encodings of instructions refer to section 8.2 Instruction Encodings. Section 8.2 explains the general structure of instruction encodings, and defines exactly the encodings of all fields contained within the instruction.

Instruction Clock Count Assumptions

1. The instruction has been prefetched, decoded, and is ready for execution.

2. Bus cycles do not require wait states.
3. There are no local bus HOLD requests delaying processor access to the bus.
4. No exceptions are detected during instruction execution.
5. If an effective address is calculated, it does not use two general register components. One register, scaling and displacement can be used within the clock counts shown. However, if the effective address calculation uses two general register components, add 1 clock to the clock count shown.

Instruction Clock Count Notation

1. If two clock counts are given, the smaller refers to a register operand and the larger refers to a memory operand.
2. n = number of times repeated.
3. m = number of components in the next instruction executed, where the entire displacement (if any) counts as one component, the entire immediate data (if any) counts as one component, and each of the other bytes of the instruction and prefix(es) each count as one component.

This appendix is an excerpt from Section 8 of the 80386 Data Sheet.8. Reprinted courtesy of Intel Corporation, Santa Clara, California.

80386 Programming Guide

INSTRUCTION	FORMAT	CLOCK COUNT		NOTES				
		Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode	Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode			
GENERAL DATA TRANSFER								
MOV = Move:								
Register to Register/Memory	<table><tr><td>1 0 0 0 1 0 0 w</td><td>mod reg</td><td>r/m</td></tr></table>	1 0 0 0 1 0 0 w	mod reg	r/m	2/2	2/2	b h	
1 0 0 0 1 0 0 w	mod reg	r/m						
Register/Memory to Register	<table><tr><td>1 0 0 0 1 0 1 w</td><td>mod reg</td><td>r/m</td></tr></table>	1 0 0 0 1 0 1 w	mod reg	r/m	2/4	2/4	b h	
1 0 0 0 1 0 1 w	mod reg	r/m						
Immediate to Register/Memory	<table><tr><td>1 1 0 0 0 1 1 w</td><td>mod 0 0 0</td><td>r/m</td></tr></table> immediate data	1 1 0 0 0 1 1 w	mod 0 0 0	r/m	2/2	2/2	b h	
1 1 0 0 0 1 1 w	mod 0 0 0	r/m						
Immediate to Register (short form)	<table><tr><td>1 0 1 1 w</td><td>reg</td></tr></table> immediate data	1 0 1 1 w	reg	2	2			
1 0 1 1 w	reg							
Memory to Accumulator (short form)	<table><tr><td>1 0 1 0 0 0 0 w</td></tr></table> full displacement	1 0 1 0 0 0 0 w	4	4	b h			
1 0 1 0 0 0 0 w								
Accumulator to Memory (short form)	<table><tr><td>1 0 1 0 0 0 1 w</td></tr></table> full displacement	1 0 1 0 0 0 1 w	2	2	b h			
1 0 1 0 0 0 1 w								
Register Memory to Segment Register	<table><tr><td>1 0 0 0 1 1 1 0</td><td>mod sreg3</td><td>r/m</td></tr></table>	1 0 0 0 1 1 1 0	mod sreg3	r/m	2/5	18/19	b h, i, j	
1 0 0 0 1 1 1 0	mod sreg3	r/m						
Segment Register to Register/Memory	<table><tr><td>1 0 0 0 1 1 0 0</td><td>mod sreg3</td><td>r/m</td></tr></table>	1 0 0 0 1 1 0 0	mod sreg3	r/m	2/2	2/2	b h	
1 0 0 0 1 1 0 0	mod sreg3	r/m						
MOVSX = Move With Sign Extension								
Register From Register/Memory	<table><tr><td>0 0 0 0 1 1 1 1</td><td>1 0 1 1 1 1 1 w</td><td>mod reg</td><td>r/m</td></tr></table>	0 0 0 0 1 1 1 1	1 0 1 1 1 1 1 w	mod reg	r/m	3/6	3/6	b h
0 0 0 0 1 1 1 1	1 0 1 1 1 1 1 w	mod reg	r/m					
MOVZX = Move With Zero Extension								
Register From Register/Memory	<table><tr><td>0 0 0 0 1 1 1 1</td><td>1 0 1 1 0 1 1 w</td><td>mod reg</td><td>r/m</td></tr></table>	0 0 0 0 1 1 1 1	1 0 1 1 0 1 1 w	mod reg	r/m	3/6	3/6	b h
0 0 0 0 1 1 1 1	1 0 1 1 0 1 1 w	mod reg	r/m					
PUSH = Push:								
Register/Memory	<table><tr><td>1 1 1 1 1 1 1 1</td><td>mod 1 1 0</td><td>r/m</td></tr></table>	1 1 1 1 1 1 1 1	mod 1 1 0	r/m	5	5	b h	
1 1 1 1 1 1 1 1	mod 1 1 0	r/m						
Register (short form)	<table><tr><td>0 1 0 1 0</td><td>reg</td></tr></table>	0 1 0 1 0	reg	2	2	b h		
0 1 0 1 0	reg							
Segment Register (ES, CS, SS or DS) (short form)	<table><tr><td>0 0 0 sreg2</td><td>1 1 0</td></tr></table>	0 0 0 sreg2	1 1 0	2	2	b h		
0 0 0 sreg2	1 1 0							
Segment Register (ES, CS, SS, DS, FS or GS)	<table><tr><td>0 0 0 0 1 1 1 1</td><td>1 0 sreg3</td><td>0 0 0</td></tr></table>	0 0 0 0 1 1 1 1	1 0 sreg3	0 0 0	2	2	b h	
0 0 0 0 1 1 1 1	1 0 sreg3	0 0 0						
Immediate	<table><tr><td>0 1 1 0 1 0 s</td><td>0</td></tr></table> immediate data	0 1 1 0 1 0 s	0	2	2	b h		
0 1 1 0 1 0 s	0							
PUSHA = Push All	<table><tr><td>0 1 1 0 0 0 0 0</td></tr></table>	0 1 1 0 0 0 0 0	18	18	b h			
0 1 1 0 0 0 0 0								
POP = Pop								
Register/Memory	<table><tr><td>1 0 0 0 1 1 1 1</td><td>mod 0 0 0</td><td>r/m</td></tr></table>	1 0 0 0 1 1 1 1	mod 0 0 0	r/m	5	5	b h	
1 0 0 0 1 1 1 1	mod 0 0 0	r/m						
Register (short form)	<table><tr><td>0 1 0 1 1</td><td>reg</td></tr></table>	0 1 0 1 1	reg	4	4	b h		
0 1 0 1 1	reg							
Segment Register (ES, CS, SS or DS) (short form)	<table><tr><td>0 0 0 sreg2</td><td>1 1 1</td></tr></table>	0 0 0 sreg2	1 1 1	7	21	b h, i, j		
0 0 0 sreg2	1 1 1							
Segment Register (ES, CS, SS or DS FS or GS)	<table><tr><td>0 0 0 0 1 1 1 1</td><td>1 0 sreg3</td><td>0 0 1</td></tr></table>	0 0 0 0 1 1 1 1	1 0 sreg3	0 0 1	7	21	b h, i, j	
0 0 0 0 1 1 1 1	1 0 sreg3	0 0 1						
POPA = Pop All	<table><tr><td>0 1 1 0 0 0 0 1</td></tr></table>	0 1 1 0 0 0 0 1	24	24	b h			
0 1 1 0 0 0 0 1								
XCHG = Exchange								
Register/Memory With Register	<table><tr><td>1 0 0 0 0 1 1 w</td><td>mod reg</td><td>r/m</td></tr></table>	1 0 0 0 0 1 1 w	mod reg	r/m	3/5	3/5	b, f f, h	
1 0 0 0 0 1 1 w	mod reg	r/m						
Register With Accumulator (short form)	<table><tr><td>1 0 0 1 0</td><td>reg</td></tr></table>	1 0 0 1 0	reg	3	3			
1 0 0 1 0	reg							
IN = Input from:		Clk Count Virtual 8086 Mode						
Fixed Port	<table><tr><td>1 1 1 0 0 1 0 w</td><td>port number</td></tr></table>	1 1 1 0 0 1 0 w	port number	†26	12	6*/26**	m	
1 1 1 0 0 1 0 w	port number							
Variable Port	<table><tr><td>1 1 1 0 1 1 0 w</td></tr></table>	1 1 1 0 1 1 0 w	†27	13	7*/27**	m		
1 1 1 0 1 1 0 w								
OUT = Output to:								
Fixed Port	<table><tr><td>1 1 1 0 0 1 1 w</td><td>port number</td></tr></table>	1 1 1 0 0 1 1 w	port number	†24	10	4*/24**	m	
1 1 1 0 0 1 1 w	port number							
Variable Port	<table><tr><td>1 1 1 0 1 1 1 w</td></tr></table>	1 1 1 0 1 1 1 w	†25	11	5*/25**	m		
1 1 1 0 1 1 1 w								
LEA = Load EA to Register	<table><tr><td>1 0 0 0 1 1 0 1</td><td>mod reg</td><td>r/m</td></tr></table>	1 0 0 0 1 1 0 1	mod reg	r/m	2	2		
1 0 0 0 1 1 0 1	mod reg	r/m						
* If CPL ≤ IOPL ** If CPL > IOPL								

* If CPL ≤ IOPL

** If CPL > IOPL

Appendix A

INSTRUCTION	FORMAT	CLOCK COUNT		NOTES					
		Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode	Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode				
SEGMENT CONTROL									
LDS = Load Pointer to DS	<table><tr><td>11000101</td><td>mod reg</td><td>r/m</td></tr></table>	11000101	mod reg	r/m	7	22	b	h, i, j	
11000101	mod reg	r/m							
LES = Load Pointer to ES	<table><tr><td>11000100</td><td>mod reg</td><td>r/m</td></tr></table>	11000100	mod reg	r/m	7	22	b	h, i, j	
11000100	mod reg	r/m							
LFS = Load Pointer to FS	<table><tr><td>00001111</td><td>10110100</td><td>mod reg</td><td>r/m</td></tr></table>	00001111	10110100	mod reg	r/m	7	25	b	h, i, j
00001111	10110100	mod reg	r/m						
LGS = Load Pointer to GS	<table><tr><td>00001111</td><td>10110101</td><td>mod reg</td><td>r/m</td></tr></table>	00001111	10110101	mod reg	r/m	7	25	b	h, i, j
00001111	10110101	mod reg	r/m						
LSS = Load Pointer to SS	<table><tr><td>00001111</td><td>10110010</td><td>mod reg</td><td>r/m</td></tr></table>	00001111	10110010	mod reg	r/m	7	22	b	h, i, j
00001111	10110010	mod reg	r/m						
FLAG CONTROL									
CLC = Clear Carry Flag	<table><tr><td>11111000</td></tr></table>	11111000	2	2					
11111000									
CLD = Clear Direction Flag	<table><tr><td>11111100</td></tr></table>	11111100	2	2					
11111100									
CLI = Clear Interrupt Enable Flag	<table><tr><td>11111010</td></tr></table>	11111010	3	3		m			
11111010									
CLTS = Clear Task Switched Flag	<table><tr><td>00001111</td><td>00000110</td></tr></table>	00001111	00000110	5	5	c	l		
00001111	00000110								
CMC = Complement Carry Flag	<table><tr><td>11110101</td></tr></table>	11110101	2	2					
11110101									
LAHF = Load AH into Flag	<table><tr><td>10011111</td></tr></table>	10011111	2	2					
10011111									
POPF = Pop Flags	<table><tr><td>10011101</td></tr></table>	10011101	5	5	b	h, n			
10011101									
PUSHF = Push Flags	<table><tr><td>10011100</td></tr></table>	10011100	4	4	b	h			
10011100									
SAHF = Store AH into Flags	<table><tr><td>10011110</td></tr></table>	10011110	3	3					
10011110									
STC = Set Carry Flag	<table><tr><td>11111001</td></tr></table>	11111001	2	2					
11111001									
STD = Set Direction Flag	<table><tr><td>11111001</td></tr></table>	11111001	2	2					
11111001									
STI = Set Interrupt Enable Flag	<table><tr><td>11111011</td></tr></table>	11111011	3	3		m			
11111011									
ARITHMETIC									
ADD = Add									
Register to Register	<table><tr><td>000000dw</td><td>mod reg</td><td>r/m</td></tr></table>	000000dw	mod reg	r/m	2	2			
000000dw	mod reg	r/m							
Register to Memory	<table><tr><td>0000000w</td><td>mod reg</td><td>r/m</td></tr></table>	0000000w	mod reg	r/m	7	7	b	h	
0000000w	mod reg	r/m							
Memory to Register	<table><tr><td>0000001w</td><td>mod reg</td><td>r/m</td></tr></table>	0000001w	mod reg	r/m	6	6	b	h	
0000001w	mod reg	r/m							
Immediate to Register/Memory	<table><tr><td>100000sw</td><td>mod 000</td><td>r/m</td><td>immediate data</td></tr></table>	100000sw	mod 000	r/m	immediate data	2/7	2/7	b	h
100000sw	mod 000	r/m	immediate data						
Immediate to Accumulator (short form)	<table><tr><td>0000010w</td><td>immediate data</td></tr></table>	0000010w	immediate data	2	2				
0000010w	immediate data								
ADC = Add With Carry									
Register to Register	<table><tr><td>000100dw</td><td>mod reg</td><td>r/m</td></tr></table>	000100dw	mod reg	r/m	2	2			
000100dw	mod reg	r/m							
Register to Memory	<table><tr><td>0001000w</td><td>mod reg</td><td>r/m</td></tr></table>	0001000w	mod reg	r/m	7	7	b	h	
0001000w	mod reg	r/m							
Memory to Register	<table><tr><td>0001001w</td><td>mod reg</td><td>r/m</td></tr></table>	0001001w	mod reg	r/m	6	6	b	h	
0001001w	mod reg	r/m							
Immediate to Register/Memory	<table><tr><td>100000sw</td><td>mod 010</td><td>r/m</td><td>immediate data</td></tr></table>	100000sw	mod 010	r/m	immediate data	2/7	2/7	b	h
100000sw	mod 010	r/m	immediate data						
Immediate to Accumulator (short form)	<table><tr><td>0001010w</td><td>immediate data</td></tr></table>	0001010w	immediate data	2	2				
0001010w	immediate data								
INC = Increment									
Register/Memory	<table><tr><td>1111111w</td><td>mod 000</td><td>r/m</td></tr></table>	1111111w	mod 000	r/m	2/6	2/6	b	h	
1111111w	mod 000	r/m							
Register (short form)	<table><tr><td>01000</td><td>reg</td></tr></table>	01000	reg	2	2				
01000	reg								
SUB = Subtract									
Register from Register	<table><tr><td>001010dw</td><td>mod reg</td><td>r/m</td></tr></table>	001010dw	mod reg	r/m	2	2			
001010dw	mod reg	r/m							

80386 Programming Guide

		CLOCK COUNT		NOTES					
INSTRUCTION	FORMAT	Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode	Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode				
ARITHMETIC (Continued)									
Register from Memory	<table><tr><td>0010100w</td><td>mod reg</td><td>r/m</td></tr></table>	0010100w	mod reg	r/m	7	7	b	h	
0010100w	mod reg	r/m							
Memory from Register	<table><tr><td>0010101w</td><td>mod reg</td><td>r/m</td></tr></table>	0010101w	mod reg	r/m	6	6	b	h	
0010101w	mod reg	r/m							
Immediate from Register/Memory	<table><tr><td>100000sw</td><td>mod 101</td><td>r/m</td></tr></table> immediate data	100000sw	mod 101	r/m	2/7	2/7	b	h	
100000sw	mod 101	r/m							
Immediate from Accumulator (short form)	<table><tr><td>0010110w</td></tr></table> immediate data	0010110w	2	2					
0010110w									
SBB = Subtract with Borrow									
Register from Register	<table><tr><td>000110dw</td><td>mod reg</td><td>r/m</td></tr></table>	000110dw	mod reg	r/m	2	2			
000110dw	mod reg	r/m							
Register from Memory	<table><tr><td>0001100w</td><td>mod reg</td><td>r/m</td></tr></table>	0001100w	mod reg	r/m	7	7	b	h	
0001100w	mod reg	r/m							
Memory from Register	<table><tr><td>0001101w</td><td>mod reg</td><td>r/m</td></tr></table>	0001101w	mod reg	r/m	6	6	b	h	
0001101w	mod reg	r/m							
Immediate from Register/Memory	<table><tr><td>100000sw</td><td>mod 011</td><td>r/m</td></tr></table> immediate data	100000sw	mod 011	r/m	2/7	2/7	b	h	
100000sw	mod 011	r/m							
Immediate from Accumulator (short form)	<table><tr><td>0001110w</td></tr></table> immediate data	0001110w	2	2					
0001110w									
DEC = Decrement									
Register/Memory	<table><tr><td>1111111w</td><td>reg 001</td><td>r/m</td></tr></table>	1111111w	reg 001	r/m	2/6	2/6	b	n	
1111111w	reg 001	r/m							
Register (short form)	<table><tr><td>01001</td><td>reg</td></tr></table>	01001	reg	2	2				
01001	reg								
CMP = Compare									
Register with Register	<table><tr><td>001110dw</td><td>mod reg</td><td>r/m</td></tr></table>	001110dw	mod reg	r/m	2	2			
001110dw	mod reg	r/m							
Memory with Register	<table><tr><td>0011100w</td><td>mod reg</td><td>r/m</td></tr></table>	0011100w	mod reg	r/m	5	5	b	h	
0011100w	mod reg	r/m							
Register with Memory	<table><tr><td>0011101w</td><td>mod reg</td><td>r/m</td></tr></table>	0011101w	mod reg	r/m	6	6	b	h	
0011101w	mod reg	r/m							
Immediate with Register/Memory	<table><tr><td>100000sw</td><td>mod 111</td><td>r/m</td></tr></table> immediate data	100000sw	mod 111	r/m	2/5	2/5	b	h	
100000sw	mod 111	r/m							
Immediate with Accumulator (short form)	<table><tr><td>0011110w</td></tr></table> immediate data	0011110w	2	2					
0011110w									
NEG = Change Sign	<table><tr><td>1111011w</td><td>mod 011</td><td>r/m</td></tr></table>	1111011w	mod 011	r/m	2/6	2/6	b	h	
1111011w	mod 011	r/m							
AAA = ASCII Adjust for Add	<table><tr><td>00110111</td></tr></table>	00110111	4	4					
00110111									
AAS = ASCII Adjust for Subtract	<table><tr><td>00111111</td></tr></table>	00111111	4	4					
00111111									
DAA = Decimal Adjust for Add	<table><tr><td>00100111</td></tr></table>	00100111	4	4					
00100111									
DAS = Decimal Adjust for Subtract	<table><tr><td>00101111</td></tr></table>	00101111	4	4					
00101111									
MUL = Multiply (unsigned)									
Accumulator with Register/Memory	<table><tr><td>1111011w</td><td>mod 100</td><td>r/m</td></tr></table>	1111011w	mod 100	r/m					
1111011w	mod 100	r/m							
Multiplier-Byte		9-14/12-17	9-14/12-17	b, d	d, h				
-Word		9-22/12-25	9-22/12-25	b, d	d, h				
-Doubleword		9-38/12-41	9-38/12-41	b, d	d, h				
IMUL = Integer Multiply (signed)									
Accumulator with Register/Memory	<table><tr><td>1111011w</td><td>mod 100</td><td>r/m</td></tr></table>	1111011w	mod 100	r/m					
1111011w	mod 100	r/m							
Multiplier-Byte		9-14/12-17	9-14/12-17	b, d	d, h				
-Word		9-22/12-25	9-22/12-25	b, d	d, h				
-Doubleword		9-38/12-41	9-38/12-41	b, d	d, h				
Register with Register/Memory	<table><tr><td>00001111</td><td>10101111</td><td>mod reg</td><td>r/m</td></tr></table>	00001111	10101111	mod reg	r/m				
00001111	10101111	mod reg	r/m						
-Word		9-22/12-25	9-22/12-25	b, d	d, h				
-Doubleword		9-38/12-41	9-38/12-41	b, d	d, h				
Register/Memory with Immediate to Register	<table><tr><td>011010s1</td><td>mod reg</td><td>r/m</td></tr></table> immediate data	011010s1	mod reg	r/m					
011010s1	mod reg	r/m							
-Word		9-22/12-25	9-22/12-25	b, d	d, h				
-Doubleword		9-38/12-41	9-38/12-41	b, d	d, h				

Appendix A

INSTRUCTION	FORMAT	CLOCK COUNT		NOTES	
		Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode	Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode
ARITHMETIC (Continued)					
DIV = Divide (Unsigned)					
Accumulator by Register/Memory	1 1 1 1 0 1 1 w mod 1 1 0 r/m				
Divisor—Byte		14/17	14/17	b,e	e,h
—Word		22/25	22/25	b,e	e,h
—Doubleword		38/41	38/41	b,e	e,h
IDIV = Integer Divide (Signed)					
Accumulator By Register/Memory	1 1 1 1 0 1 1 w mod 1 1 1 r/m				
Divisor—Byte		19/22	19/22	b,e	e,h
—Word		27/30	27/30	b,e	e,h
—Doubleword		43/46	43/46	b,e	e,h
AAD = ASCII Adjust for Divide	1 1 0 1 0 1 0 1 0 0 0 0 1 0 1 0	19	19		
AAM = ASCII Adjust for Multiply	1 1 0 1 0 1 0 0 0 0 0 1 0 1 0	17	17		
CBW = Convert Byte to Word	1 0 0 1 1 0 0 0	3	3		
CWD = Convert Word to Double Word	1 0 0 1 1 0 0 1	2	2		
LOGIC					
Shift Rotate Instructions					
Not Through Carry (ROL, ROR, SAL, SAR, SHL, and SHR)					
Register/Memory by 1	1 1 0 1 0 0 0 w mod TTT r/m	3/7	3/7	b	h
Register/Memory by CL	1 1 0 1 0 0 1 w mod TTT r/m	3/7	3/7	b	h
Register/Memory by Immediate Count	1 1 0 0 0 0 0 w mod TTT r/m	3/7	3/7	b	h
immed 8-bit data					
Through Carry (RCL and RCR)					
Register/Memory by 1	1 1 0 1 0 0 0 w mod TTT r/m	9/10	9/10	b	h
Register/Memory by CL	1 1 0 1 0 0 1 w mod TTT r/m	9/10	9/10	b	h
Register/Memory by Immediate Count	1 1 0 0 0 0 0 w mod TTT r/m	9/10	9/10	b	h
immed 8-bit data					
		TTT Instruction			
		0 0 0 ROL			
		0 0 1 ROR			
		0 1 0 RCL			
		0 1 1 RCR			
		1 0 0 SHL/SAL			
		1 0 1 SHR			
		1 1 1 SAR			
SHLD = Shift Left Double					
Register/Memory by Immediate	0 0 0 0 1 1 1 1 1 0 1 0 0 1 0 0 mod reg r/m	3/7	3/7		
immed 8-bit data					
Register/Memory by CL	0 0 0 0 1 1 1 1 1 0 1 0 0 1 0 1 mod reg r/m	3/7	3/7		
SHRD = Shift Right Double					
Register/Memory by Immediate	0 0 0 0 1 1 1 1 1 0 1 0 1 1 0 0 mod reg r/m	3/7	3/7		
immed 8-bit data					
Register/Memory by CL	0 0 0 0 1 1 1 1 1 0 1 0 1 1 0 1 mod reg r/m	3/7	3/7		
AND = And					
Register to Register	0 0 1 0 0 0 d w mod reg r/m	2	2		

80386 Programming Guide

INSTRUCTION	FORMAT	CLOCK COUNT		NOTES		
		Real Address Mode or Virtual 8088 Mode	Protected Virtual Address Mode	Real Address Mode or Virtual 8088 Mode	Protected Virtual Address Mode	
LOGIC (Continued)						
Register to Memory	<div>0 0 1 0 0 0 0 w</div> <div>mod reg r/m</div>	7	7	b	h	
Memory to Register	<div>0 0 1 0 0 0 1 w</div> <div>mod reg r/m</div>	6	6	b	h	
Immediate to Register/Memory	<div>1 0 0 0 0 0 0 w</div> <div>mod 1 0 0 r/m</div> immediate data	2/7	2/7	b	h	
Immediate to Accumulator (Short Form)	<div>0 0 1 0 0 1 0 w</div> immediate data	2	2			
TEST = And Function to Flags, No Result						
Register/Memory and Register	<div>1 0 0 0 0 1 0 w</div> <div>mod reg r/m</div>	2/5	2/5	b	h	
Immediate Data and Register/Memory	<div>1 1 1 1 0 1 1 w</div> <div>mod 0 0 0 r/m</div> immediate data	2/5	2/5	b	h	
Immediate Data and Accumulator (Short Form)	<div>1 0 1 0 1 0 0 w</div> immediate data	2	2			
OR = Or						
Register to Register	<div>0 0 0 0 1 0 d w</div> <div>mod reg r/m</div>	2	2			
Register to Memory	<div>0 0 0 0 1 0 0 w</div> <div>mod reg r/m</div>	7	7	b	h	
Memory to Register	<div>0 0 0 0 1 0 1 w</div> <div>mod reg r/m</div>	6	6	b	h	
Immediate to Register/Memory	<div>1 0 0 0 0 0 0 w</div> <div>mod 0 0 1 r/m</div> immediate data	2/7	2/7	b	h	
Immediate to Accumulator (Short Form)	<div>0 0 0 0 1 1 0 w</div> immediate data	2	2			
XOR = Exclusive Or						
Register to Register	<div>0 0 1 1 0 0 d w</div> <div>mod reg r/m</div>	2	2			
Register to Memory	<div>0 0 1 1 0 0 0 w</div> <div>mod reg r/m</div>	7	7	b	h	
Memory to Register	<div>0 0 1 1 0 0 1 w</div> <div>mod reg r/m</div>	6	6	b	h	
Immediate to Register/Memory	<div>1 0 0 0 0 0 0 w</div> <div>mod 1 1 0 r/m</div> immediate data	2/7	2/7	b	h	
Immediate to Accumulator (Short Form)	<div>0 0 1 1 0 1 0 w</div> immediate data	2	2			
NOT = Invert Register/Memory	<div>1 1 1 1 0 1 1 w</div> <div>mod 0 1 0 r/m</div>	2/6	2/6	b	h	
STRING MANIPULATION						
CMPS = Compare Byte Word	<div>1 0 1 0 0 1 1 w</div>	Cik Count Virtual 8086 Mode	10	10	b	h
INS = Input Byte/Word from DX Port	<div>0 1 1 0 1 1 0 w</div>		15	9*/29**	b	h, m
LODS = Load Byte/Word to AL/AX/EAX	<div>1 0 1 0 1 1 0 w</div>		5	5	b	h
MOVS = Move Byte Word	<div>1 0 1 0 0 1 0 w</div>		7	7	b	h
OUTS = Output Byte/Word to DX Port	<div>0 1 1 0 1 1 1 w</div>		14	8*/28**	b	h, m
SCAS = Scan Byte Word	<div>1 0 1 0 1 1 1 w</div>		7	7	b	h
STOS = Store Byte/Word from AL/AX/EX	<div>1 0 1 0 1 0 1 w</div>		4	4	b	h
XLAT = Translate String	<div>1 1 0 1 0 1 1 1</div>		5	5		h
REPEATED STRING MANIPULATION						
Repeated by Count in CX or ECX						
REPE CMPS = Compare String (Find Non-Match)	<div>1 1 1 1 0 0 1 1</div> <div>1 0 1 0 0 1 1 w</div>		5+9n	5+9n	b	h

* If CPL ≤ IOPL

** If CPL > IOPL

Cik
Count
Virtual
8086
Mode

Appendix A

INSTRUCTION	FORMAT		CLOCK COUNT		NOTES						
			Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode	Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode					
REPEATED STRING MANIPULATION (Continued)											
REPNE CMPS = Compare String (Find Match)	<table><tr><td>11110010</td><td>1010011w</td></tr></table>	11110010	1010011w	Clk Count Virtual 8086 Mode	5+9n	5+9n	b	h			
11110010	1010011w										
REP INS = Input String	<table><tr><td>11110010</td><td>0110110w</td></tr></table>	11110010	0110110w	†27+6n	13+6n	7+6n*/27+6n**	b	h, m			
11110010	0110110w										
REP LODS = Load String	<table><tr><td>11110010</td><td>1010110w</td></tr></table>	11110010	1010110w		5+6n	5+6n	b	h			
11110010	1010110w										
REP MOVS = Move String	<table><tr><td>11110010</td><td>1010010w</td></tr></table>	11110010	1010010w		7+4n	7+4n	b	h			
11110010	1010010w										
REP OUTS = Output String	<table><tr><td>11110010</td><td>0110111w</td></tr></table>	11110010	0110111w	†26+5n	12+5n	6+5n*/26+5n**	b	h, m			
11110010	0110111w										
REPE SCAS = Scan String (Find Non-AL/AX/EAX)	<table><tr><td>11110011</td><td>1010111w</td></tr></table>	11110011	1010111w		5+8n	5+8n	b	h			
11110011	1010111w										
REPNE SCAS = Scan String (Find AL/AX/EAX)	<table><tr><td>11110010</td><td>1010111w</td></tr></table>	11110010	1010111w		5+8n	5+8n	b	h			
11110010	1010111w										
REP STOS = Store String	<table><tr><td>11110010</td><td>1010101w</td></tr></table>	11110010	1010101w		5+5n	5+5n	b	h			
11110010	1010101w										
BIT MANIPULATION											
BSF = Scan Bit Forward	<table><tr><td>00001111</td><td>10111100</td><td>mod reg</td><td>r/m</td></tr></table>	00001111	10111100	mod reg	r/m		10+3n	10+3n	b	h	
00001111	10111100	mod reg	r/m								
BSR = Scan Bit Reverse	<table><tr><td>00001111</td><td>10111101</td><td>mod reg</td><td>r/m</td></tr></table>	00001111	10111101	mod reg	r/m		10+3n	10+3n	b	h	
00001111	10111101	mod reg	r/m								
BT = Test Bit											
Register/Memory, Immediate	<table><tr><td>00001111</td><td>10111010</td><td>mod 100</td><td>r/m</td><td>immed 8-bit data</td></tr></table>	00001111	10111010	mod 100	r/m	immed 8-bit data		3/6	3/6	b	h
00001111	10111010	mod 100	r/m	immed 8-bit data							
Register/Memory, Register	<table><tr><td>00001111</td><td>10100011</td><td>mod reg</td><td>r/m</td></tr></table>	00001111	10100011	mod reg	r/m		3/12	3/12	b	h	
00001111	10100011	mod reg	r/m								
BTC = Test Bit and Complement											
Register/Memory, Immediate	<table><tr><td>00001111</td><td>10111010</td><td>mod 111</td><td>r/m</td><td>immed 8-bit data</td></tr></table>	00001111	10111010	mod 111	r/m	immed 8-bit data		6/8	6/8	b	h
00001111	10111010	mod 111	r/m	immed 8-bit data							
Register/Memory, Register	<table><tr><td>00001111</td><td>10111011</td><td>mod reg</td><td>r/m</td></tr></table>	00001111	10111011	mod reg	r/m		6/13	6/13	b	h	
00001111	10111011	mod reg	r/m								
BTR = Test Bit and Reset											
Register/Memory, Immediate	<table><tr><td>00001111</td><td>10111010</td><td>mod 110</td><td>r/m</td><td>immed 8-bit data</td></tr></table>	00001111	10111010	mod 110	r/m	immed 8-bit data		6/8	6/8	b	h
00001111	10111010	mod 110	r/m	immed 8-bit data							
Register/Memory, Register	<table><tr><td>00001111</td><td>10110011</td><td>mod reg</td><td>r/m</td></tr></table>	00001111	10110011	mod reg	r/m		6/13	6/13	b	h	
00001111	10110011	mod reg	r/m								
BTS = Test Bit and Set											
Register/Memory, Immediate	<table><tr><td>00001111</td><td>10111010</td><td>mod 101</td><td>r/m</td><td>immed 8-bit data</td></tr></table>	00001111	10111010	mod 101	r/m	immed 8-bit data		6/8	6/8	b	h
00001111	10111010	mod 101	r/m	immed 8-bit data							
Register/Memory, Register	<table><tr><td>00001111</td><td>10101011</td><td>mod reg</td><td>r/m</td></tr></table>	00001111	10101011	mod reg	r/m		6/13	6/13	b	h	
00001111	10101011	mod reg	r/m								
CONTROL TRANSFER											
CALL = Call											
Direct Within Segment	<table><tr><td>11101000</td><td>full displacement</td></tr></table>	11101000	full displacement		7+m	7+m	b	r			
11101000	full displacement										
Register/Memory											
Indirect Within Segment	<table><tr><td>11111111</td><td>mod 010</td><td>r/m</td></tr></table>	11111111	mod 010	r/m		7+m/ 10+m	7+m/ 10+m	b	h, r		
11111111	mod 010	r/m									
Direct Intersegment	<table><tr><td>10011010</td><td>unsigned full offset, selector</td></tr></table>	10011010	unsigned full offset, selector		17+m	34+m	b	j,k,r			
10011010	unsigned full offset, selector										

Notes:

† Clock count shown applies if I/O permission allows I/O to the port in virtual 8086 mode. If I/O bit map denies permission exception 13 fault occurs; refer to clock counts for INT 3 instruction.

* If CPL ≤ IOPL

** If CPL > IOPL

80386 Programming Guide

INSTRUCTION	FORMAT	CLOCK COUNT		NOTES				
		Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode	Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode			
CONTROL TRANSFER (Continued)								
Protected Mode Only (Direct Intersegment)								
Via Call Gate to Same Privilege Level			52 + m		h,j,k,r			
Via Call Gate to Different Privilege Level, (No Parameters)			86 + m		h,j,k,r			
Via Call Gate to Different Privilege Level, (x Parameters)			94 + 4x + m		h,j,k,r			
From 286 Task to 286 TSS			273		h,j,k,r			
From 286 Task to 386 TSS			298		h,j,k,r			
From 286 Task to Virtual 8086 Task (386 TSS)			217		h,j,k,r			
From 386 Task to 286 TSS			273		h,j,k,r			
From 386 Task to 386 TSS			300		h,j,k,r			
From 386 Task to Virtual 8086 Task (386 TSS)			217		h,j,k,r			
Indirect Intersegment	<table><tr><td>1 1 1 1 1 1 1 1</td><td>mod 0 1 1</td><td>r/m</td></tr></table>	1 1 1 1 1 1 1 1	mod 0 1 1	r/m	22 + m	38 + m	b	h,j,k,r
1 1 1 1 1 1 1 1	mod 0 1 1	r/m						
Protected Mode Only (Indirect Intersegment)								
Via Call Gate to Same Privilege Level			56 + m		h,j,k,r			
Via Call Gate to Different Privilege Level, (No Parameters)			90 + m		h,j,k,r			
Via Call Gate to Different Privilege Level, (x Parameters)			98 + 4x + m		h,j,k,r			
From 286 Task to 286 TSS			278		h,j,k,r			
From 286 Task to 386 TSS			303		h,j,k,r			
From 286 Task to Virtual 8086 Task (386 TSS)			221		h,j,k,r			
From 386 Task to 286 TSS			278		h,j,k,r			
From 386 Task to 386 TSS			305		h,j,k,r			
From 386 Task to Virtual 8086 Task (386 TSS)			221		h,j,k,r			
JMP = Unconditional Jump								
Short	<table><tr><td>1 1 1 0 1 0 0 1</td><td>8-bit displacement</td></tr></table>	1 1 1 0 1 0 0 1	8-bit displacement	7 + m	7 + m		r	
1 1 1 0 1 0 0 1	8-bit displacement							
Direct within Segment	<table><tr><td>1 1 1 0 1 0 0 1</td><td>full displacement</td></tr></table>	1 1 1 0 1 0 0 1	full displacement	7 + m	7 + m		r	
1 1 1 0 1 0 0 1	full displacement							
Register/Memory Indirect within Segment	<table><tr><td>1 1 1 1 1 1 1 1</td><td>mod 1 0 0</td><td>r/m</td></tr></table>	1 1 1 1 1 1 1 1	mod 1 0 0	r/m	7 + m/ 10 + m	7 + m/ 10 + m	b	h,r
1 1 1 1 1 1 1 1	mod 1 0 0	r/m						
Direct Intersegment	<table><tr><td>1 1 1 0 1 0 1 0</td><td>unsigned full offset, selector</td></tr></table>	1 1 1 0 1 0 1 0	unsigned full offset, selector	12 + m	27 + m		j,k,r	
1 1 1 0 1 0 1 0	unsigned full offset, selector							
Protected Mode Only (Direct Intersegment)								
Via Call Gate to Same Privilege Level			45 + m		h,j,k,r			
From 286 Task to 286 TSS			274		h,j,k,r			
From 286 Task to 386 TSS			301		h,j,k,r			
From 286 Task to Virtual 8086 Task (386 TSS)			218		h,j,k,r			
From 386 Task to 286 TSS			270		h,j,k,r			
From 386 Task to 386 TSS			303		h,j,k,r			
From 386 Task to Virtual 8086 Task (386 TSS)			220		h,j,k,r			
Indirect Intersegment	<table><tr><td>1 1 1 1 1 1 1 1</td><td>mod 1 0 1</td><td>r/m</td></tr></table>	1 1 1 1 1 1 1 1	mod 1 0 1	r/m	17 + m	31 + m	b	h,j,k,r
1 1 1 1 1 1 1 1	mod 1 0 1	r/m						
Protected Mode Only (Indirect Intersegment)								
Via Call Gate to Same Privilege Level			49 + m		h,j,k,r			
From 286 Task to 286 TSS			279		h,j,k,r			
From 286 Task to 386 TSS			306		h,j,k,r			
From 286 Task to Virtual 8086 Task (386 TSS)			222		h,j,k,r			
From 386 Task to 286 TSS			275		h,j,k,r			
From 386 Task to 386 TSS			308		h,j,k,r			
From 386 Task to Virtual 8086 Task (386 TSS)			224		h,j,k,r			

Appendix A

INSTRUCTION	FORMAT	CLOCK COUNT		NOTES				
		Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode	Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode			
CONTROL TRANSFER (Continued)								
RET = Return from CALL:								
Within Segment	<table><tr><td>1 1 0 0 0 0 1 1</td><td></td></tr></table>	1 1 0 0 0 0 1 1		10 + m	10 + m	b	g, h, r	
1 1 0 0 0 0 1 1								
Within Segment Adding Immediate to SP	<table><tr><td>1 1 0 0 0 0 1 0</td><td>16-bit displ</td></tr></table>	1 1 0 0 0 0 1 0	16-bit displ	10 + m	10 + m	b	g, h, r	
1 1 0 0 0 0 1 0	16-bit displ							
Intersegment	<table><tr><td>1 1 0 0 1 0 1 1</td><td></td></tr></table>	1 1 0 0 1 0 1 1		18 + m	32+m	b	g, h, j, k, r	
1 1 0 0 1 0 1 1								
Intersegment Adding Immediate to SP	<table><tr><td>1 1 0 0 1 0 1 0</td><td>16-bit displ</td></tr></table>	1 1 0 0 1 0 1 0	16-bit displ	18 + m	32+m	b	g, h, j, k, r	
1 1 0 0 1 0 1 0	16-bit displ							
Protected Mode Only (RET):								
to Different Privilege Level								
Intersegment			68		h, j, k, r			
Intersegment Adding Immediate to SP			68		h, j, k, r			
CONDITIONAL JUMPS								
NOTE: Times Are Jump "Taken or Not Taken"								
JO = Jump on Overflow								
8-Bit Displacement	<table><tr><td>0 1 1 1 0 0 0 0</td><td>8-bit displ</td></tr></table>	0 1 1 1 0 0 0 0	8-bit displ	7 + m or 3	7 + m or 3		r	
0 1 1 1 0 0 0 0	8-bit displ							
Full Displacement	<table><tr><td>0 0 0 0 1 1 1 1</td><td>1 0 0 0 0 0 0 0</td><td>full displacement</td></tr></table>	0 0 0 0 1 1 1 1	1 0 0 0 0 0 0 0	full displacement	7 + m or 3	7 + m or 3		r
0 0 0 0 1 1 1 1	1 0 0 0 0 0 0 0	full displacement						
JNO = Jump on Not Overflow								
8-Bit Displacement	<table><tr><td>0 1 1 1 0 0 0 1</td><td>8-bit displ</td></tr></table>	0 1 1 1 0 0 0 1	8-bit displ	7 + m or 3	7 + m or 3		r	
0 1 1 1 0 0 0 1	8-bit displ							
Full Displacement	<table><tr><td>0 0 0 0 1 1 1 1</td><td>1 0 0 0 0 0 0 1</td><td>full displacement</td></tr></table>	0 0 0 0 1 1 1 1	1 0 0 0 0 0 0 1	full displacement	7 + m or 3	7 + m or 3		r
0 0 0 0 1 1 1 1	1 0 0 0 0 0 0 1	full displacement						
JB/JNAE = Jump on Below/Not Above or Equal								
8-Bit Displacement	<table><tr><td>0 1 1 1 0 0 1 0</td><td>8-bit displ</td></tr></table>	0 1 1 1 0 0 1 0	8-bit displ	7 + m or 3	7 + m or 3		r	
0 1 1 1 0 0 1 0	8-bit displ							
Full Displacement	<table><tr><td>0 0 0 0 1 1 1 1</td><td>1 0 0 0 0 0 1 0</td><td>full displacement</td></tr></table>	0 0 0 0 1 1 1 1	1 0 0 0 0 0 1 0	full displacement	7 + m or 3	7 + m or 3		r
0 0 0 0 1 1 1 1	1 0 0 0 0 0 1 0	full displacement						
JNB/JAE = Jump on Not Below/Above or Equal								
8-Bit Displacement	<table><tr><td>0 1 1 1 0 0 1 1</td><td>8-bit displ</td></tr></table>	0 1 1 1 0 0 1 1	8-bit displ	7 + m or 3	7 + m or 3		r	
0 1 1 1 0 0 1 1	8-bit displ							
Full Displacement	<table><tr><td>0 0 0 0 1 1 1 1</td><td>1 0 0 0 0 0 1 1</td><td>full displacement</td></tr></table>	0 0 0 0 1 1 1 1	1 0 0 0 0 0 1 1	full displacement	7 + m or 3	7 + m or 3		r
0 0 0 0 1 1 1 1	1 0 0 0 0 0 1 1	full displacement						
JE/JZ = Jump on Equal/Zero								
8-Bit Displacement	<table><tr><td>0 1 1 1 0 1 0 0</td><td>8-bit displ</td></tr></table>	0 1 1 1 0 1 0 0	8-bit displ	7 + m or 3	7 + m or 3		r	
0 1 1 1 0 1 0 0	8-bit displ							
Full Displacement	<table><tr><td>0 0 0 0 1 1 1 1</td><td>1 0 0 0 0 1 0 0</td><td>full displacement</td></tr></table>	0 0 0 0 1 1 1 1	1 0 0 0 0 1 0 0	full displacement	7 + m or 3	7 + m or 3		r
0 0 0 0 1 1 1 1	1 0 0 0 0 1 0 0	full displacement						
JNE/JNZ = Jump on Not Equal/Not Zero								
8-Bit Displacement	<table><tr><td>0 1 1 1 0 1 0 1</td><td>8-bit displ</td></tr></table>	0 1 1 1 0 1 0 1	8-bit displ	7 + m or 3	7 + m or 3		r	
0 1 1 1 0 1 0 1	8-bit displ							
Full Displacement	<table><tr><td>0 0 0 0 1 1 1 1</td><td>1 0 0 0 0 1 0 1</td><td>full displacement</td></tr></table>	0 0 0 0 1 1 1 1	1 0 0 0 0 1 0 1	full displacement	7 + m or 3	7 + m or 3		r
0 0 0 0 1 1 1 1	1 0 0 0 0 1 0 1	full displacement						
JBE/JNA = Jump on Below or Equal/Not Above								
8-Bit Displacement	<table><tr><td>0 1 1 1 0 1 1 0</td><td>8-bit displ</td></tr></table>	0 1 1 1 0 1 1 0	8-bit displ	7 + m or 3	7 + m or 3		r	
0 1 1 1 0 1 1 0	8-bit displ							
Full Displacement	<table><tr><td>0 0 0 0 1 1 1 1</td><td>1 0 0 0 0 1 1 0</td><td>full displacement</td></tr></table>	0 0 0 0 1 1 1 1	1 0 0 0 0 1 1 0	full displacement	7 + m or 3	7 + m or 3		r
0 0 0 0 1 1 1 1	1 0 0 0 0 1 1 0	full displacement						
JNBE/JA = Jump on Not Below or Equal/Above								
8-Bit Displacement	<table><tr><td>0 1 1 1 0 1 1 1</td><td>8-bit displ</td></tr></table>	0 1 1 1 0 1 1 1	8-bit displ	7 + m or 3	7 + m or 3		r	
0 1 1 1 0 1 1 1	8-bit displ							
Full Displacement	<table><tr><td>0 0 0 0 1 1 1 1</td><td>1 0 0 0 0 1 1 1</td><td>full displacement</td></tr></table>	0 0 0 0 1 1 1 1	1 0 0 0 0 1 1 1	full displacement	7 + m or 3	7 + m or 3		r
0 0 0 0 1 1 1 1	1 0 0 0 0 1 1 1	full displacement						
JS = Jump on Sign								
8-Bit Displacement	<table><tr><td>0 1 1 1 1 0 0 0</td><td>8-bit displ</td></tr></table>	0 1 1 1 1 0 0 0	8-bit displ	7 + m or 3	7 + m or 3		r	
0 1 1 1 1 0 0 0	8-bit displ							
Full Displacement	<table><tr><td>0 0 0 0 1 1 1 1</td><td>1 0 0 0 1 0 0 0</td><td>full displacement</td></tr></table>	0 0 0 0 1 1 1 1	1 0 0 0 1 0 0 0	full displacement	7 + m or 3	7 + m or 3		r
0 0 0 0 1 1 1 1	1 0 0 0 1 0 0 0	full displacement						

80386 Programming Guide

INSTRUCTION	FORMAT	CLOCK COUNT		NOTES					
		Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode	Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode				
CONDITIONAL JUMPS (Continued)									
JNS = Jump on Not Sign									
8-Bit Displacement	<table><tr><td>0 1 1 1 0 0 1</td><td>8-bit displ</td></tr></table>	0 1 1 1 0 0 1	8-bit displ	7 + m or 3	7 + m or 3		r		
0 1 1 1 0 0 1	8-bit displ								
Full Displacement	<table><tr><td>0 0 0 0 1 1 1</td><td>1 0 0 0 1 0 0 1</td><td>full displacement</td></tr></table>	0 0 0 0 1 1 1	1 0 0 0 1 0 0 1	full displacement	7 + m or 3	7 + m or 3		r	
0 0 0 0 1 1 1	1 0 0 0 1 0 0 1	full displacement							
JP/JPE = Jump on Parity/Parity Even									
8-Bit Displacement	<table><tr><td>0 1 1 1 0 1 0</td><td>8-bit displ</td></tr></table>	0 1 1 1 0 1 0	8-bit displ	7 + m or 3	7 + m or 3		r		
0 1 1 1 0 1 0	8-bit displ								
Full Displacement	<table><tr><td>0 0 0 0 1 1 1</td><td>1 0 0 0 1 0 1 0</td><td>full displacement</td></tr></table>	0 0 0 0 1 1 1	1 0 0 0 1 0 1 0	full displacement	7 + m or 3	7 + m or 3		r	
0 0 0 0 1 1 1	1 0 0 0 1 0 1 0	full displacement							
JNP/JPO = Jump on Not Parity/Parity Odd									
8-Bit Displacement	<table><tr><td>0 1 1 1 0 1 1</td><td>8-bit displ</td></tr></table>	0 1 1 1 0 1 1	8-bit displ	7 + m or 3	7 + m or 3		r		
0 1 1 1 0 1 1	8-bit displ								
Full Displacement	<table><tr><td>0 0 0 0 1 1 1</td><td>1 0 0 0 1 0 1 1</td><td>full displacement</td></tr></table>	0 0 0 0 1 1 1	1 0 0 0 1 0 1 1	full displacement	7 + m or 3	7 + m or 3		r	
0 0 0 0 1 1 1	1 0 0 0 1 0 1 1	full displacement							
JL/JNGE = Jump on Less/Not Greater or Equal									
8-Bit Displacement	<table><tr><td>0 1 1 1 1 0 0</td><td>8-bit displ</td></tr></table>	0 1 1 1 1 0 0	8-bit displ	7 + m or 3	7 + m or 3		r		
0 1 1 1 1 0 0	8-bit displ								
Full Displacement	<table><tr><td>0 0 0 0 1 1 1</td><td>1 0 0 0 1 1 0 0</td><td>full displacement</td></tr></table>	0 0 0 0 1 1 1	1 0 0 0 1 1 0 0	full displacement	7 + m or 3	7 + m or 3		r	
0 0 0 0 1 1 1	1 0 0 0 1 1 0 0	full displacement							
JNL/JGE = Jump on Not Less/Greater or Equal									
8-Bit Displacement	<table><tr><td>0 1 1 1 1 0 1</td><td>8-bit displ</td></tr></table>	0 1 1 1 1 0 1	8-bit displ	7 + m or 3	7 + m or 3		r		
0 1 1 1 1 0 1	8-bit displ								
Full Displacement	<table><tr><td>0 0 0 0 1 1 1</td><td>1 0 0 0 1 1 0 1</td><td>full displacement</td></tr></table>	0 0 0 0 1 1 1	1 0 0 0 1 1 0 1	full displacement	7 + m or 3	7 + m or 3		r	
0 0 0 0 1 1 1	1 0 0 0 1 1 0 1	full displacement							
JLE/JNG = Jump on Less or Equal/Not Greater									
8-Bit Displacement	<table><tr><td>0 1 1 1 1 1 0</td><td>8-bit displ</td></tr></table>	0 1 1 1 1 1 0	8-bit displ	7 + m or 3	7 + m or 3		r		
0 1 1 1 1 1 0	8-bit displ								
Full Displacement	<table><tr><td>0 0 0 0 1 1 1</td><td>1 0 0 0 1 1 1 0</td><td>full displacement</td></tr></table>	0 0 0 0 1 1 1	1 0 0 0 1 1 1 0	full displacement	7 + m or 3	7 + m or 3		r	
0 0 0 0 1 1 1	1 0 0 0 1 1 1 0	full displacement							
JNLE/JG = Jump on Not Less or Equal/Greater									
8-Bit Displacement	<table><tr><td>0 1 1 1 1 1 1</td><td>8-bit displ</td></tr></table>	0 1 1 1 1 1 1	8-bit displ	7 + m or 3	7 + m or 3		r		
0 1 1 1 1 1 1	8-bit displ								
Full Displacement	<table><tr><td>0 0 0 0 1 1 1</td><td>1 0 0 0 1 1 1 1</td><td>full displacement</td></tr></table>	0 0 0 0 1 1 1	1 0 0 0 1 1 1 1	full displacement	7 + m or 3	7 + m or 3		r	
0 0 0 0 1 1 1	1 0 0 0 1 1 1 1	full displacement							
JCXZ = Jump on CX Zero									
	<table><tr><td>1 1 1 0 0 0 1 1</td><td>8-bit displ</td></tr></table>	1 1 1 0 0 0 1 1	8-bit displ	9 + m or 5	9 + m or 5		r		
1 1 1 0 0 0 1 1	8-bit displ								
JECXZ = Jump on ECX Zero									
	<table><tr><td>1 1 1 0 0 0 1 1</td><td>8-bit displ</td></tr></table>	1 1 1 0 0 0 1 1	8-bit displ	9 + m or 5	9 + m or 5		r		
1 1 1 0 0 0 1 1	8-bit displ								
(Address Size Prefix Differentiates JCXZ from JECXZ)									
LOOP = Loop CX Times									
	<table><tr><td>1 1 1 0 0 0 1 0</td><td>8-bit displ</td></tr></table>	1 1 1 0 0 0 1 0	8-bit displ	11 + m	11 + m		r		
1 1 1 0 0 0 1 0	8-bit displ								
LOOPZ/LOOPE = Loop with Zero/Equal									
	<table><tr><td>1 1 1 0 0 0 0 1</td><td>8-bit displ</td></tr></table>	1 1 1 0 0 0 0 1	8-bit displ	11 + m	11 + m		r		
1 1 1 0 0 0 0 1	8-bit displ								
LOOPNZ/LOOPNE = Loop While Not Zero									
	<table><tr><td>1 1 1 0 0 0 0 0</td><td>8-bit displ</td></tr></table>	1 1 1 0 0 0 0 0	8-bit displ	11 + m	11 + m		r		
1 1 1 0 0 0 0 0	8-bit displ								
CONDITIONAL BYTE SET									
NOTE: Times Are Register/Memory									
SETO = Set Byte on Overflow									
To Register/Memory	<table><tr><td>0 0 0 0 1 1 1 1</td><td>1 0 0 1 0 0 0 0</td><td>mod 0 0 0</td><td>r/m</td></tr></table>	0 0 0 0 1 1 1 1	1 0 0 1 0 0 0 0	mod 0 0 0	r/m	4/5	4/5		h
0 0 0 0 1 1 1 1	1 0 0 1 0 0 0 0	mod 0 0 0	r/m						
SETNO = Set Byte on Not Overflow									
To Register/Memory	<table><tr><td>0 0 0 0 1 1 1 1</td><td>1 0 0 1 0 0 0 1</td><td>mod 0 0 0</td><td>r/m</td></tr></table>	0 0 0 0 1 1 1 1	1 0 0 1 0 0 0 1	mod 0 0 0	r/m	4/5	4/5		h
0 0 0 0 1 1 1 1	1 0 0 1 0 0 0 1	mod 0 0 0	r/m						
SETB/SETNAE = Set Byte on Below/Not Above or Equal									
To Register/Memory	<table><tr><td>0 0 0 0 1 1 1 1</td><td>1 0 0 1 0 0 1 0</td><td>mod 0 0 0</td><td>r/m</td></tr></table>	0 0 0 0 1 1 1 1	1 0 0 1 0 0 1 0	mod 0 0 0	r/m	4/5	4/5		h
0 0 0 0 1 1 1 1	1 0 0 1 0 0 1 0	mod 0 0 0	r/m						

Appendix A

INSTRUCTION	FORMAT	CLOCK COUNT		NOTES					
		Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode	Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode				
CONDITIONAL BYTE SET (Continued)									
SETNB = Set Byte on Not Below/Above or Equal									
To Register/Memory	<table><tr><td>00001111</td><td>10010011</td><td>mod000</td><td>r/m</td></tr></table>	00001111	10010011	mod000	r/m	4/5	4/5		h
00001111	10010011	mod000	r/m						
SETE/SETZ = Set Byte on Equal/Zero									
To Register/Memory	<table><tr><td>00001111</td><td>10010100</td><td>mod000</td><td>r/m</td></tr></table>	00001111	10010100	mod000	r/m	4/5	4/5		h
00001111	10010100	mod000	r/m						
SETNE/SETNZ = Set Byte on Not Equal/Not Zero									
To Register/Memory	<table><tr><td>00001111</td><td>10010101</td><td>mod000</td><td>r/m</td></tr></table>	00001111	10010101	mod000	r/m	4/5	4/5		h
00001111	10010101	mod000	r/m						
SETBE/SETNA = Set Byte on Below or Equal/Not Above									
To Register/Memory	<table><tr><td>00001111</td><td>10010110</td><td>mod000</td><td>r/m</td></tr></table>	00001111	10010110	mod000	r/m	4/5	4/5		h
00001111	10010110	mod000	r/m						
SETNBE/SETA = Set Byte on Not Below or Equal/Above									
To Register/Memory	<table><tr><td>00001111</td><td>10010111</td><td>mod000</td><td>r/m</td></tr></table>	00001111	10010111	mod000	r/m	4/5	4/5		h
00001111	10010111	mod000	r/m						
SETS = Set Byte on Sign									
To Register/Memory	<table><tr><td>00001111</td><td>10011000</td><td>mod000</td><td>r/m</td></tr></table>	00001111	10011000	mod000	r/m	4/5	4/5		h
00001111	10011000	mod000	r/m						
SETNS = Set Byte on Not Sign									
To Register/Memory	<table><tr><td>00001111</td><td>10011001</td><td>mod000</td><td>r/m</td></tr></table>	00001111	10011001	mod000	r/m	4/5	4/5		h
00001111	10011001	mod000	r/m						
SETP/SETPE = Set Byte on Parity/Parity Even									
To Register/Memory	<table><tr><td>00001111</td><td>10011010</td><td>mod000</td><td>r/m</td></tr></table>	00001111	10011010	mod000	r/m	4/5	4/5		h
00001111	10011010	mod000	r/m						
SETNP/SETPO = Set Byte on Not Parity/Parity Odd									
To Register/Memory	<table><tr><td>00001111</td><td>10011011</td><td>mod000</td><td>r/m</td></tr></table>	00001111	10011011	mod000	r/m	4/5	4/5		h
00001111	10011011	mod000	r/m						
SETL/SETNGE = Set Byte on Less/Not Greater or Equal									
To Register/Memory	<table><tr><td>00001111</td><td>10011100</td><td>mod000</td><td>r/m</td></tr></table>	00001111	10011100	mod000	r/m	4/5	4/5		h
00001111	10011100	mod000	r/m						
SETNL/SETGE = Set Byte on Not Less/Greater or Equal									
To Register/Memory	<table><tr><td>00001111</td><td>01111101</td><td>mod000</td><td>r/m</td></tr></table>	00001111	01111101	mod000	r/m	4/5	4/5		h
00001111	01111101	mod000	r/m						
SETLE/SETNG = Set Byte on Less or Equal/Not Greater									
To Register/Memory	<table><tr><td>00001111</td><td>10011110</td><td>mod000</td><td>r/m</td></tr></table>	00001111	10011110	mod000	r/m	4/5	4/5		h
00001111	10011110	mod000	r/m						
SETNLE/SETG = Set Byte on Not Less or Equal/Greater									
To Register/Memory	<table><tr><td>00001111</td><td>10011111</td><td>mod000</td><td>r/m</td></tr></table>	00001111	10011111	mod000	r/m	4/5	4/5		h
00001111	10011111	mod000	r/m						
ENTER = Enter Procedure									
	<table><tr><td>11001000</td><td colspan="3">16-bit displacement, 8-bit level</td></tr></table>	11001000	16-bit displacement, 8-bit level						
11001000	16-bit displacement, 8-bit level								
L = 0		10	10	b	h				
L = 1		12	12	b	h				
L > 1		15 + 4(n - 1)	15 + 4(n - 1)	b	h				
LEAVE = Leave Procedure									
	<table><tr><td>11001001</td></tr></table>	11001001	4	4	b	h			
11001001									

80386 Programming Guide

INSTRUCTION	FORMAT	CLOCK COUNT		NOTES			
		Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode	Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode		
INTERRUPT INSTRUCTIONS							
INT = Interrupt:							
Type Specified	<table><tr><td>1 1 0 0 1 1 0 1</td><td>type</td></tr></table>	1 1 0 0 1 1 0 1	type	37		b	
1 1 0 0 1 1 0 1	type						
Type 3	<table><tr><td>1 1 0 0 1 1 0 0</td></tr></table>	1 1 0 0 1 1 0 0	33		b		
1 1 0 0 1 1 0 0							
INTO = Interrupt 4 If Overflow Flag Set							
	<table><tr><td>1 1 0 0 1 1 1 0</td></tr></table>	1 1 0 0 1 1 1 0					
1 1 0 0 1 1 1 0							
If OF = 1		35		b, e			
If OF = 0		3	3	b, e			
Bound = Interrupt 5 if Detect Value Out of Range							
	<table><tr><td>0 1 1 0 0 0 1 0</td><td>mod reg</td><td>r/m</td></tr></table>	0 1 1 0 0 0 1 0	mod reg	r/m			
0 1 1 0 0 0 1 0	mod reg	r/m					
If Out of Range		44		b, e	e, g, h, j, k, r		
If In Range		10	10	b, e	e, g, h, j, k, r		
Protected Mode Only (INT)							
INT: Type Specified							
Via Interrupt or Trap Gate to Same Privilege Level							
			59		g, j, k, r		
Via Interrupt or Trap Gate to Different Privilege Level							
			99		g, j, k, r		
From 286 Task to 286 TSS via Task Gate							
			282		g, j, k, r		
From 286 Task to 386 TSS via Task Gate							
			309		g, j, k, r		
From 286 Task to virt 8086 md via Task Gate							
			226		g, j, k, r		
From 386 Task to 286 TSS via Task Gate							
			284		g, j, k, r		
From 386 Task to 386 TSS via Task Gate							
			311		g, j, k, r		
From 386 Task to virt 8086 md via Task Gate							
			228		g, j, k, r		
From virt 8086 md to 286 TSS via Task Gate							
			289		g, j, k, r		
From virt 8086 md to 386 TSS via Task Gate							
			316		g, j, k, r		
From virt 8086 md to priv level 0 via Trap Gate or Interrupt Gate							
			119				
INT: TYPE 3							
Via Interrupt or Trap Gate to Same Privilege Level							
			59		*g, j, k, r		
Via Interrupt or Trap Gate to Different Privilege Level							
			99		g, j, k, r		
From 286 Task to 286 TSS via Task Gate							
			278		g, j, k, r		
From 286 Task to 386 TSS via Task Gate							
			305		g, j, k, r		
From 286 Task to Virt 8086 md via Task Gate							
			222		g, j, k, r		
From 386 Task to 286 TSS via Task Gate							
			280		g, j, k, r		
From 386 Task to 386 TSS via Task Gate							
			307		g, j, k, r		
From 386 Task to Virt 8086 md via Task Gate							
			224		g, j, k, r		
From virt 8086 md to 286 TSS via Task Gate							
			285		g, j, k, r		
From virt 8086 md to 386 TSS via Task Gate							
			312		g, j, k, r		
From virt 8086 md to priv level 0 via Trap Gate or Interrupt Gate							
			119				
INTO:							
Via Interrupt or Trap Gate to Same Privilege Level							
			59		g, j, k, r		
Via Interrupt or Trap Gate to Different Privilege Level							
			99		g, j, k, r		
From 286 Task to 286 TSS via Task Gate							
			280		g, j, k, r		
From 286 Task to 386 TSS via Task Gate							
			307		g, j, k, r		
From 286 Task to virt 8086 md via Task Gate							
			224		g, j, k, r		
From 386 Task to 286 TSS via Task Gate							
			282		g, j, k, r		
From 386 Task to 386 TSS via Task Gate							
			309		g, j, k, r		
From 386 Task to virt 8086 md via Task Gate							
			226		g, j, k, r		
From virt 8086 md to 286 TSS via Task Gate							
			287		g, j, k, r		
From virt 8086 md to 386 TSS via Task Gate							
			314		g, j, k, r		
From virt 8086 md to priv level 0 via Trap Gate or Interrupt Gate							
			119				

Appendix A

INSTRUCTION	FORMAT	CLOCK COUNT		NOTES	
		Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode	Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode
INTERRUPT INSTRUCTIONS (Continued)					
BOUND:					
Via Interrupt or Trap Gate to Same Privilege Level			59		g, j, k, r
Via Interrupt or Trap Gate to Different Privilege Level			99		g, j, k, r
From 286 Task to 286 TSS via Task Gate			254		g, j, k, r
From 286 Task to 386 TSS via Task Gate			284		g, j, k, r
From 286 Task to virt 8086 Mode via Task Gate			231		g, j, k, r
From 386 Task to 286 TSS via Task Gate			264		g, j, k, r
From 386 Task to 386 TSS via Task Gate			294		g, j, k, r
From 386 Task to virt 8086 Mode via Task Gate			243		g, j, k, r
From virt 8086 Mode to 286 TSS via Task Gate			264		g, j, k, r
From virt 8086 Mode to 386 TSS via Task Gate			294		g, j, k, r
From virt 8086 md to priv level 0 via Trap Gate or Interrupt Gate			119		
INTERRUPT RETURN					
IRET = Interrupt Return	11001111	22			g, h, j, k, r
Protected Mode Only (IRET)					
To the Same Privilege Level (within task)			38		g, h, j, k, r
To Different Privilege Level (within task)			82		g, h, j, k, r
From 286 Task to 286 TSS			232		h, j, k, r
From 286 Task to 386 TSS			265		h, j, k, r
From 286 Task to Virtual 8086 Task			214		h, j, k, r
From 286 Task to Virtual 8086 Mode (within task)			60		
From 386 Task to 286 TSS			271		h, j, k, r
From 386 Task to 386 TSS			275		h, j, k, r
From 386 Task to Virtual 8086 Task			224		h, j, k, r
From 386 Task to Virtual 8086 Mode (within task)			60		
PROCESSOR CONTROL					
HLT = HALT	11110100	5	5		I
MOV = Move to and From Control/Debug/Test Registers					
CR0/CR2/CR3 from register	00001111 00100010 11eee reg	10/4/5	10/4/5		I
Register From CR0-3	00001111 00100000 11eee reg	6	6		I
DR0-3 From Register	00001111 00100011 11eee reg	22	22		I
DR6-7 From Register	00001111 00100011 11eee reg	16	16		I
Register from DR6-7	00001111 00100001 11eee reg	14	14		I
Register from DR0-3	00001111 00100001 11eee reg	22	22		I
TR6-7 from Register	00001111 00100110 11eee reg	12	12		I
Register from TR6-7	00001111 00100100 11eee reg	12	12		I
NOP = No Operation	10010000	3	3		
WAIT = Wait until BUSY# pin is negated	10011011	6	6		

80386 Programming Guide

INSTRUCTION		FORMAT	CLOCK COUNT		NOTES				
			Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode	Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode			
PROCESSOR EXTENSION INSTRUCTIONS									
Processor Extension Escape	<table><tr><td>11011TTT</td><td>mod LLL</td><td>r/m</td></tr></table> TTT and LLL bits are opcode information for coprocessor.	11011TTT	mod LLL	r/m	See 80287/80387 data sheets for clock counts			h	
11011TTT	mod LLL	r/m							
PREFIX BYTES									
Address Size Prefix	<table><tr><td>01100111</td></tr></table>	01100111	0	0					
01100111									
LOCK = Bus Lock Prefix	<table><tr><td>11110000</td></tr></table>	11110000	0	0		m			
11110000									
Operand Size Prefix	<table><tr><td>01100110</td></tr></table>	01100110	0	0					
01100110									
Segment Override Prefix									
CS:	<table><tr><td>00101110</td></tr></table>	00101110	0	0					
00101110									
DS:	<table><tr><td>00111110</td></tr></table>	00111110	0	0					
00111110									
ES:	<table><tr><td>00100110</td></tr></table>	00100110	0	0					
00100110									
FS:	<table><tr><td>01100100</td></tr></table>	01100100	0	0					
01100100									
GS:	<table><tr><td>01100101</td></tr></table>	01100101	0	0					
01100101									
SS:	<table><tr><td>00110110</td></tr></table>	00110110	0	0					
00110110									
PROTECTION CONTROL									
ARPL = Adjust Requested Privilege Level									
From Register/Memory	<table><tr><td>01100011</td><td>mod reg</td><td>r/m</td></tr></table>	01100011	mod reg	r/m	N/A	20/21	a	h	
01100011	mod reg	r/m							
LAR = Load Access Rights									
From Register/Memory	<table><tr><td>00001111</td><td>00000010</td><td>mod reg</td><td>r/m</td></tr></table>	00001111	00000010	mod reg	r/m	N/A	15/16	a	g, h, i, p
00001111	00000010	mod reg	r/m						
LGDT = Load Global Descriptor									
Table Register	<table><tr><td>00001111</td><td>00000001</td><td>mod 010</td><td>r/m</td></tr></table>	00001111	00000001	mod 010	r/m	11	11	b, c	h, i
00001111	00000001	mod 010	r/m						
LIDT = Load Interrupt Descriptor									
Table Register	<table><tr><td>00001111</td><td>00000001</td><td>mod 011</td><td>r/m</td></tr></table>	00001111	00000001	mod 011	r/m	11	11	b, c	h, i
00001111	00000001	mod 011	r/m						
LLDT = Load Local Descriptor									
Table Register to Register/Memory	<table><tr><td>00001111</td><td>00000000</td><td>mod 010</td><td>r/m</td></tr></table>	00001111	00000000	mod 010	r/m	N/A	20/24	a	g, h, i, l
00001111	00000000	mod 010	r/m						
LMSW = Load Machine Status Word									
From Register/Memory	<table><tr><td>00001111</td><td>00000001</td><td>mod 110</td><td>r/m</td></tr></table>	00001111	00000001	mod 110	r/m	10/13	10/13	b, c	h, i
00001111	00000001	mod 110	r/m						
LSL = Load Segment Limit									
From Register/Memory	<table><tr><td>00001111</td><td>00000011</td><td>mod reg</td><td>r/m</td></tr></table>	00001111	00000011	mod reg	r/m				
00001111	00000011	mod reg	r/m						
	Byte-Granular Limit	N/A	20/21	a	g, h, i, p				
	Page-Granular Limit	N/A	25/26	a	g, h, i, p				
LTR = Load Task Register									
From Register/Memory	<table><tr><td>00001111</td><td>00000000</td><td>mod 001</td><td>r/m</td></tr></table>	00001111	00000000	mod 001	r/m	N/A	23/27	a	g, h, i, l
00001111	00000000	mod 001	r/m						
SGDT = Store Global Descriptor									
Table Register	<table><tr><td>00001111</td><td>00000001</td><td>mod 000</td><td>r/m</td></tr></table>	00001111	00000001	mod 000	r/m	9	9	b, c	h
00001111	00000001	mod 000	r/m						

Appendix A

INSTRUCTION	FORMAT	CLOCK COUNT		NOTES	
		Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode	Real Address Mode or Virtual 8086 Mode	Protected Virtual Address Mode
SIDT	= Store Interrupt Descriptor Table Register				
	To Register/Memory	00001111 00000001 mod 001 r/m	9	9	b, c h
SLOT	= Store Local Descriptor Table Register				
	To Register/Memory	00001111 00000000 mod 000 r/m	N/A	2/2	a h
SMSW	= Store Machine Status Word				
	To Register/Memory	00001111 00000001 mod 100 r/m	10/13	10/13	b, c h, i
STR	= Store Task Register				
	To Register/Memory	00001111 00000000 mod 001 r/m	N/A	2/2	a h
VERR	= Verify Read Access				
	Register/Memory	00001111 00000000 mod 100 r/m	N/A	10/11	a g, h, j, p
VERW	= Verify Write Access				
	Register/Memory	00001111 00000000 mod 101 r/m	N/A	15/16	a g, h, j, p

INSTRUCTION NOTES FOR TABLE A-1

Notes a through c apply to 80386 Real Address Mode only:

- This is a Protected Mode instruction. Attempted execution in Real Mode will result in exception 6 (invalid opcode).
- Exception 13 fault (general protection) will occur in Real Mode if an operand reference is made that partially or fully extends beyond the maximum CS, DS, ES, FS or GS limit, FFFFH. Exception 12 fault (stack segment limit violation or not present) will occur in Real Mode if an operand reference is made that partially or fully extends beyond the maximum SS limit.
- This instruction may be executed in Real Mode. In Real Mode, its purpose is primarily to initialize the CPU for Protected Mode.

Notes d through g apply to 80386 Real Address Mode and 80386 Protected Virtual Address Mode:

- The 80386 uses an early-out multiply algorithm. The actual number of clocks depends on the position of the most significant bit in the operand (multiplier).

Clock counts given are minimum to maximum. To calculate actual clocks use the following formula:

Actual Clock = if $m \neq 0$ then $\max(\lceil \log_2 |m| \rceil, 3) + 6$ clocks:

Actual Clock = if $m = 0$ then 9 clocks (where m is the multiplier)

- An exception may occur, depending on the value of the operand.
- LOCK# is automatically asserted, regardless of the presence or absence of the LOCK# prefix.
- LOCK# is asserted during descriptor table accesses.

80386 Programming Guide

Notes h through r apply to 80386 Protected Virtual Address Mode only:

- h. Exception 13 fault (general protection violation) will occur if the memory operand in CS, DS, ES, FS or GS cannot be used due to either a segment limit violation or access rights violation. If a stack limit is violated, an exception 12 (stack segment limit violation or not present) occurs.
- j. All segment descriptor access in the GDT or LDT made by this instruction will automatically assert LOCK# to maintain descriptor integrity in multi-processor systems.
- k. JMP, CALL, INT, RET and IRET instructions referring to another code segment will cause an exception 13 (general protection violation) if an applicable privilege rule is violated.
- l. An exception 13 fault occurs if CPL is greater than 0 (0 is the most privileged level).
- m. An exception 13 fault occurs if CPL is greater than IOPL.
- n. The IF bit of the flag register is not updated if CPL is greater than IOPL. The IOPL and VM fields of the flag register are updated only if CPL = 0.
- o. The PE bit of the MSW (CRO) cannot be reset by this instruction. Use MOV into CRO if desiring to reset the PE bit.
- p. Any violation of privilege rules as applied to the selector operand does not cause a protection exception; rather, the zero flag is cleared.
- q. If the coprocessor's memory operand violates a segment limit or segment access rights, an exception 13 fault (general protection exception) will occur before the ESC instruction is executed. An exception 12 fault (stack segment limit violation or not present) will occur if the stack limit is violated by the operand's starting address.
- r. The destination of a JMP, CALL, INT, RET or IRET must be in the defined limit of a code segment

or an exception 13 fault (general protection violation) will occur.

INSTRUCTION ENCODING

OVERVIEW

All instruction encodings are subsets of the general instruction format shown in Figure 8-1. Instructions consist of one or two primary opcode bytes, possibly an address specifier consisting of the "mod r/m" byte and "scaled index" byte, a displacement if required, and an immediate data field if required.

Within the primary opcode or opcodes, smaller encoding fields may be defined. These fields vary according to the class of operation. The fields define such information as direction of the operation, size of the displacements, register encoding, or sign extension.

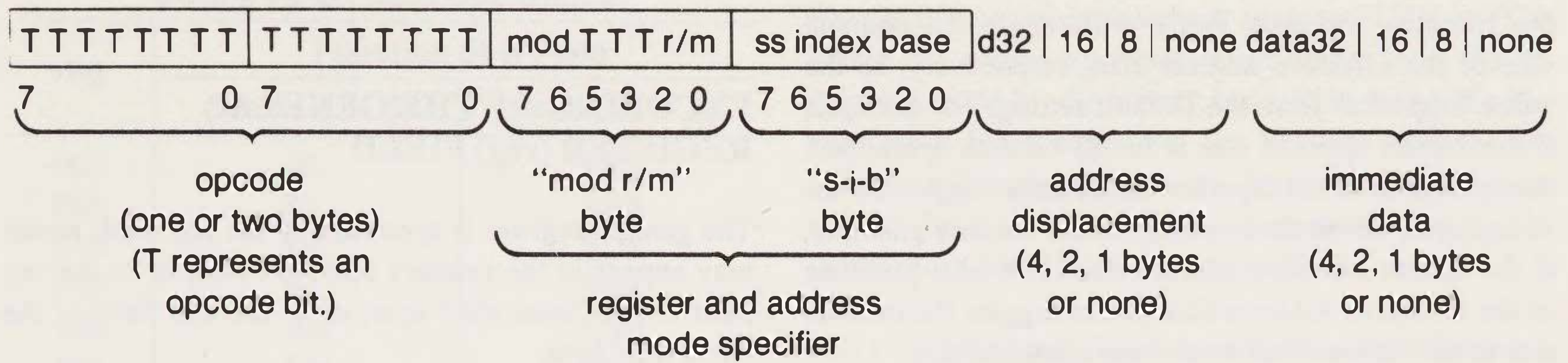
Almost all instructions referring to an operand in memory have an addressing mode byte following the primary opcode byte(s). This byte, the mod r/m byte, specifies the address mode to be used. Certain encodings of the mod r/m byte indicate a second addressing byte, the scale-index-base byte, follows the mod r/m byte to fully specify the addressing mode.

Addressing modes can include a displacement immediately following the mod r/m byte, or scaled index byte. If a displacement is present, the possible sizes are 8, 16, or 32 bits.

If the instruction specifies an immediate operand, the immediate operand follows any displacement bytes. The immediate operand, if specified, is always the last field of the instruction.

Figure A-1 illustrates several of the fields that can appear in an instruction, such as the mod field and the r/m field, but the Figure does not show all fields. Several smaller fields also appear in certain instructions, sometimes within the opcode bytes themselves. Table A-2 is a complete list of all fields appearing in the 80386 instruction set. Further ahead, following Table A-2, are detailed tables for each field.

Appendix A



Field Name	Description	Number of Bits
w	Specifies if Data is Byte or Full Size (Full Size is either 16 or 32 Bits)	1
d	Specifies Direction of Data Operation	1
s	Specifies if an Immediate Data Field Must be Sign-Extended	1
reg	General Register Specifier	3
mod r/m	Address Mode Specifier (Effective Address can be a General Register)	2 for mod; 3 for r/m
ss	Scale Factor for Scaled Index Address Mode	2
index	General Register to be used as Index Register	3
base	General Register to be used as Base Register	3
sreg2	Segment Register Specifier for CS, SS, DS, ES	2
sreg3	Segment Register Specifier for CS, SS, DS, ES, FS, GS	3
tttn	For Conditional Instructions, Specifies a Condition Asserted or a Condition Negated	4

32-BIT EXTENSIONS OF THE INSTRUCTION SET

With the 80386, the 86/186/286 instruction set is extended in two orthogonal directions: 32-bit forms of all 16-bit instructions are added to support the 32-bit data types, and 32-bit addressing modes are made available for all instructions referencing memory. This orthogonal instruction set extension is accomplished having a Default (D) bit in the code segment descriptor, and by having 2 prefixes to the instruction set.

Whether the instruction defaults to operations of 16 bits or 32 bits depends on the setting of the D bit in the code

segment descriptor, which gives the default length (either 32 bits or 16 bits) for both operands and effective addresses when executing that code segment. In the Real Address Mode or Virtual 8086 Mode, no code segment descriptors are used, but a D value of 0 is assumed internally by the 80386 when operating in those modes (for 16-bit default sizes compatible with the 8086/80186/80286).

Two prefixes, the Operand Size Prefix and the Effective Address Size Prefix, allow overriding individually the Default selection of operand size and effective address size. These prefixes may precede any opcode bytes and affect only the instruction they precede. If necessary,

80386 Programming Guide

one or both of the prefixes may be placed before the opcode bytes. The presence of the Operand Size Prefix and the Effective Address Prefix will toggle the operand size or the effective address size, respectively, to the value "opposite" from the Default setting. For example, if the default operand size is for 32-bit data operations, then presence of the Operand Size Prefix toggles the instruction to 16-bit data operation. As another example, if the default effective address size is 16 bits, presence of the Effective Address Size prefix toggles the instruction to use 32-bit effective address computations.

These 32-bit extensions are available in all 80386 modes, including the Real Address Mode or the Virtual 8086 Mode. In these modes the default is always 16 bits, so prefixes are needed to specify 32-bit operands or addresses.

Unless specified otherwise, instructions with 8-bit and 16-bit operands do not affect the contents of the high-order bits of the extended registers.

ENCODING OF INSTRUCTION FIELDS

Within the instruction are several fields indicating register selection, addressing mode and so on. the exact encodings of these fields are defined immediately ahead.

ENCODING OF OPERAND LENGTH (w) FIELD

For any given instruction performing a data operation, the instruction is executing as a 32-bit operation or a 16-bit operation. Within the constraints of the operation size, the w field encodes the operand size as either one

byte or the full operation size, as shown in the table below.

ENCODING OF THE GENERAL REGISTER (reg) FIELD

The general register is specified by the reg field, which may appear in the primary opcode bytes, or as the reg field of the "mod r/m" byte, or as the r/m field of the "mod r/m" byte.

Encoding of reg Field When w Field Is not Present in Instruction

reg Field	Register Selected During 16-Bit Data Operations	Register Selected During 32-Bit Data Operations
000	AX	EAX
001	CX	ECX
010	DX	EDX
011	BX	EBX
100	SP	ESP
101	BP	EBP
101	SI	ESI
101	DI	EDI

Encoding of reg Field When w Field Is Present in Instruction

Register Specified by reg Field During 16-Bit Data Operations:

reg	Function of w Field	
	(when w = 0)	(when w = 1)
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Register Specified by reg Field During 32-Bit Data Operations

reg	Function of w Field	
	(when w = 0)	(when w = 1)
000	AL	EAX
001	CL	ECX
010	DL	EDX
011	BL	EBX
100	AH	ESP
101	CH	EBP
110	DH	ESI
111	BH	EDI

ENCODING OF THE SEGMENT REGISTER (sreg) FIELD

The sreg field in certain instructions is a 2-bit field allowing one of the four 80286 segment registers to be specified. The sreg field in other instructions is a 3-bit field, allowing the 80386 FS and GS segment registers to be specified.

2-Bit sreg2 Field

2-Bit sreg2 Field	Segment Register Selected
00	ES
01	CS
10	SS
11	DS

3-Bit sreg3 Field

3-Bit sreg3 Field	Segment Register Selected
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS
110	do not use
111	do not use

ENCODING OF ADDRESS MODE

Except for special instructions, such as PUSH or POP, where the addressing mode is pre-determined, the addressing mode for the current instruction is specified by addressing bytes following the primary opcode. The primary addressing byte is the "mod r/m" byte, and a second byte of addressing information, the "s-i-b" (scale-index-base) byte, can be specified.

The s-i-b byte (scale-index-base byte) is specified when using 32-bit addressing mode and the "mod r/m" byte has r/m = 100 and mod = 00, 01 or 10. When the sib byte is present, the 32-bit addressing mode is a function of the mod, as, index, and base fields.

The primary addressing byte, the "mod r/m" byte, also contains three bits (shown as TTT in Figure A-1) sometimes used as an extension of the primary opcode. The three bits, however, may also be used as a register field (reg).

When calculating an effective address, either 16-bit addressing or 32-bit addressing is used. 16-bit addressing uses 16-bit address components to calculate the effective address while 32-bit addressing uses 32-bit address components to calculate the effective address. When 16-bit addressing is used, the "mod r/m" byte is interpreted as a 16-bit addressing mode specifier. When 32-bit addressing is used, the "mod r/m" byte is interpreted as a 16-bit addressing mode specifier. When 32-bit addressing is used, the "mod r/m" byte is interpreted as a 32-bit addressing mode specifier.

Tables on the following three pages define all encodings of all 16-bit addressing modes and 32-bit addressing modes.

80386 Programming Guide

Encoding of 16-bit Address Mode with "mod r/m" Byte

mod r/m	Effective Address	mod r/m	Effective Address
00 000	DS:[BX + SI]	10 000	DS:[BX + SI + d16]
00 001	DS:[BX + DI]	10 001	DS:[BX + DI + d16]
00 010	SS:[BP + SI]	10 010	SS:[BP + SI + d16]
00 011	SS:[BP + DI]	10 011	SS:[BP + DI + d16]
00 100	DS:[SI]	10 100	DS:[SI + d16]
00 101	DS:[DI]	10 101	DS:[DI + d16]
00 110	DS:d16	10 110	SS:[BP + d16]
00 111	DS:[BX]	10 111	DS:[BX + d16]
01 000	DS:[BX + SI + d8]	11 000	register—see below
01 001	DS:[BX + DI + d8]	11 001	register—see below
01 010	SS:[BP + SI + d8]	11 010	register—see below
01 011	SS:[BP + DI + d8]	11 011	register—see below
01 100	DS:[SI + d8]	11 100	register—see below
01 101	DS:[DI + d8]	11 101	register—see below
01 110	SS:[BP + d8]	11 110	register—see below
01 111	DS:[BX + d8]	11 111	register—see below

Register Specified by r/m During 16-Bit Data Operations

mod r/m	Function of w Field	
	(when w = 0)	(when w = 1)
11 000	AL	AX
11 001	CL	CX
11 010	DL	DX
11 011	BL	BX
11 100	AH	SP
11 101	CH	BP
11 110	DH	SI
11 111	BH	DI

Register Specified by r/m During 32-Bit Data Operations

mod r/m	Function of w Field	
	(when w = 0)	(when w = 1)
11 000	AL	EAX
11 001	CL	ECX
11 010	DL	EDX
11 011	BL	EBX
11 100	AH	ESP
11 101	CH	EBP
11 110	DH	ESI
11 111	BH	EDI

Appendix A

Encoding of 32-bit Address Mode with “mod r/m” byte (no “s-i-b” byte present):

mod r/m	Effective Address	mod r/m	Effective Address
00 000	DS:[EAX]	10 000	DS:[EAX + d32]
00 001	DS:[ECX]	10 001	DS:[ECX + d32]
00 010	DS:[EDX]	10 010	DS:[EDX + d32]
00 011	DS:[EBX]	10 011	DS:[EBX + d32]
00 100	s-i-b is present	10 100	s-i-b is present
00 101	DS:d32	10 101	SS:[EBP + d32]
00 110	DS:[ESI]	10 110	DS:[ESI + d32]
00 111	DS:[EDI]	10 111	DS:[EDI + d32]
01 000	DS:[EAX + d8]	11 000	register—see below
01 001	DS:[ECX + d8]	11 001	register—see below
01 010	DS:[EDX + d8]	11 010	register—see below
01 011	DS:[EBX + d8]	11 011	register—see below
01 100	s-i-b is present	11 100	register—see below
01 101	SS:[EBP + d8]	11 101	register—see below
01 110	DS:[ESI + d8]	11 110	register—see below
01 111	DS:[EDI + d8]	11 111	register—see below

Register Specified by reg or r/m during 16-Bit Data Operations:

mod r/m	function of w field	
	(when w = 0)	(when w = 1)
11 000	AL	AX
11 001	CL	CX
11 010	DL	DX
11 011	BL	BX
11 100	AH	SP
11 101	CH	BP
11 110	DH	SI
11 111	BH	DI

Register Specified by reg or r/m during 32-Bit Data Operations:

mod r/m	function of w field	
	(when w = 0)	(when w = 1)
11 000	AL	EAX
11 001	CL	ECX
11 010	DL	EDX
11 011	BL	EBX
11 100	AH	ESP
11 101	CH	EBP
11 110	DH	ESI
11 111	BH	EDI

80386 Programming Guide

Encoding of 32-bit Address Mode ("mod r/m" byte and "s-i-b" byte present):

mod base	Effective Address	ss	Scale Factor
00 000	DS:[EAX + (scaled index)]	00	x1
00 001	DS:[ECX + (scaled index)]	01	x2
00 010	DS:[EDX + (scaled index)]	10	x4
00 011	DS:[EBX + (scaled index)]	11	x8
00 100	SS:[ESP + (scaled index)]		
00 101	DS:[d32 + (scaled index)]		
00 110	DS:[ESI + (scaled index)]		
00 111	DS:[EDI + (scaled index)]		
		index	Index Register
		000	EAX
		001	ECX
		010	EDX
		011	EBX
		100	no index reg**
		101	EBP
		110	ESI
		111	EDI
01 000	DS:[EAX + (scaled index) + d8]		
01 001	DS:[ECX + (scaled index) + d8]		
01 010	DS:[EDX + (scaled index) + d8]		
01 011	DS:[EBX + (scaled index) + d8]		
01 100	SS:[ESP + (scaled index) + d8]		
01 101	SS:[EBP + (scaled index) + d8]		
01 110	DS:[ESI + (scaled index) + d8]		
01 111	DS:[EDI + (scaled index) + d8]		
10 000	DS:[EAX + (scaled index) + d32]		
10 001	DS:[ECX + (scaled index) + d32]		
10 010	DS:[EDX + (scaled index) + d32]		
10 011	DS:[EBX + (scaled index) + d32]		
10 100	SS:[ESP + (scaled index) + d32]		
10 101	SS:[EBP + (scaled index) + d32]		
10 110	DS:[ESI + (scaled index) + d32]		
10 111	DS:[EDI + (scaled index) + d32]		

**IMPORTANT NOTE:

When index field is 100, indicating "no index register," then ss field MUST equal 00. If index is 100 and ss does not equal 00, the effective address is undefined.

NOTE:

Mod field in "mod r/m" byte; ss, index, base fields in "s-i-b" byte.

Appendix A

ENCODING OF OPERATION DIRECTION (D) FIELD

In many two-operand instructions the d field is present to indicate which operand is considered the source and which is the destination.

d	Direction of Operation
0	Register/Memory < - - Register "reg" Field Indicates Source Operand; "mod r/m" or "mod ss index base" Indicates Destination Operand
1	Register < - - Register/Memory "reg" Field Indicates Destination Operand; "mod r/m" or "mod ss index base" Indicates Source Operand

Mnemonic	Condition	tttn
O	Overflow	0000
NO	No Overflow	0001
B/NAE	Below/Not Above or Equal	0010
NB/AE	Not Below/Above or Equal	0011
E/Z	Equal/Zero	0100
NE/NZ	Not Equal/Not Zero	0101
BE/NA	Below or Equal/Not Above	0110
NBE/A	Not Below or Equal/Above	0111
S	Sign	1000
NS	Not Sign	1001
P/PE	Parity/Parity Even	1010
NP/PO	Not Parity/Parity Odd	1011
L/NGE	Less Than/Not Greater or Equal	1100
NL/GE	Not Less Than/Greater or Equal	1101
LE/NG	Less Than or Equal/Greater Than	1110
NLE/G	Not Less or Equal/Greater Than	1111

ENCODING OF SIGN-EXTEND (s) FIELD

The s field occurs primarily to instructions with immediate data fields. The s field has an effect only if the size of the immediate data is 8 bits and is being placed in a 16-bit or 32-bit destination.

s	Effect on Immediate Data8	Effect on Immediate Data 16 32
0	None	None
1	Sign-Extend Data8 to Fill 16-Bit or 32-Bit Destination	None

ENCODING OF CONDITIONAL TEST (tttn) FIELD

For the conditional instructions (conditional jumps and set on condition), tttn is encoded with n indicating to use the condition (n = 0) or its negation (n = 1), and ttt giving the condition to test.

ENCODING OF CONTROL OR DEBUG OR TEST REGISTER (eee) FIELD

For the loading and storing of the Control, Debug and Test registers.

When Interpreted as Control Register Field

eee Code	Reg Name
000	CR0
010	CR2
011	CR3

Do not use any other encoding

When Interpreted as Debug Register Field

eee Code	Reg Name
000	DR0
001	DR1
010	DR2
011	DR3
110	DR6
111	DR7

Do not use any other encoding

When Interpreted as Test Register Field

eee Code	Reg Name
110	TR6
111	TR7

Do not use any other encoding

Appendix

B

Complete 80386 Flag Cross-Reference

KEY TO CODES

T	=	instruction tests flag
M	=	instruction modifies flag (either sets or resets depending on operands)
0	=	instruction resets flag
1	=	instruction sets flag
—	=	instruction's effect on flag is undefined
R	=	instruction restores prior value of flag
blank	=	instruction does not affect flag

Instruction	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
AAA	—	—	—	TM	—	M					
AAD	—	M	M	—	M	—					
AAM	—	M	M	—	M	—					
AAS	—	—	—	TM	—	M					
ADC	M	M	M	M	M	TM					
ADD	M	M	M	M	M	M					
AND	0	M	M	—	M	0					
ARPL			M								
BOUND											
BSF/BSR	—	—	M	—	—	—					
BT/BTS/BTR/BTC	—	—	—	—	—	M					
CALL											
CBW						0					
CLC						0					
CLD								0	0		
CLI								0			
CLTS											
CMC						M					
CMP	M	M	M	M	M	M					
CMPS	M	M	M	M	M	M			T		
CWD											
DAA	—	M	M	TM	M	TM					
DAS	—	M	M	TM	M	TM					
DEC	M	M	M	M	M						
DIV	—	—	—	—	—	—					
ENTER											
ESC											
HLT											
IDIV	—	—	—	—	—	—					
IMUL	M	—	—	—	—	M					
IN											
INC	M	M	M	M	M						
INS											
INT									T		
INTO	T						0 0 R			0 0 T	
IRET	R	R	R	R	R	R		R	R		
Jcond	T	T	T		T	T					

80386 Programming Guide

Instruction	OF	SF	ZF	AF	PF	CF	TF	IF	DF	NT	RF
JCXZ											
JMP											
LAHF											
LAR			M								
LDS/LES/LSS/LFS/LGS											
LEA											
LEAVE											
LGDT/LIDT/LLDT/LMSW											
LOCK											
LODS									T		
LOOP											
LOOPE/LOOPNE			T								
LSL			M								
LTR											
MOV											
MOV control, debug	—	—	—	—	—	—					
MOVS									T		
MOVSX/MOVZX											
MUL	M	—	—	—	—	M					
NEG	M	M	M	M	M	M					
NOP											
NOT											
OR	0	M	M	—	M	0					
OUT											
OUTS									T		
POP/POPA											
POPF	R	R	R	R	R	R	R	R	R	R	
PUSH/PUSHA/PUSHF											
RCL/RCR 1	M					TM					
RCL/RCR count	—					TM					
REP/REPE/REPNE											
RET											
ROL/ROR 1	M					M					
ROL/ROR count	—					M					
SAHF		R	R	R	R	R					
SAL/SAR/SHL/SHR 1	M	M	M	—	M	M					
SAL/SAR/SHL/SHR count	—	M	M	—	M	M					
SBB	M	M	M	M	M	TM					
SCAS	M	M	M	M	M	M			T		
SET cond	T	T	T		T	T					
SGDT/SIDT/SLDT/SMSW											
SHLD/SHRD	—	M	M	—	M	M					
STC						1					
STD									1		
STI								1			
STOS									T		
STR											
SUB	M	M	M	M	M	M					
TEST	0	M	M	—	M	0					
VERR/VERRW			M								
WAIT											
XCHG											
XLAT											
XOR	0	M	M	—	M	0					

Appendix

C

80386 Status Flag Summary

STATUS FLAGS' FUNCTIONS

Bit	Name	Function
0	CF	Carry Flag — Set on high-order bit carry or borrow; cleared otherwise.
2	PF	Parity Flag — Set if low-order eight bits of result contain an even number of 1 bits; cleared otherwise.
4	AF	Adjust flag — Set on carry from or borrow to the low order four bits of AL; cleared otherwise. Used for decimal arithmetic.
6	ZF	Zero Flag — Set if result is zero; cleared otherwise.
7	SF	Sign Flag — Set equal to high-order bit of result (0 is positive, 1 if negative).
11	OF	Overflow Flag — Set if result is too large a positive number or too small a negative number (excluding sign-bit) to fit in destination operand; cleared otherwise.

KEY TO CODES

T	=	instruction tests flag
M	=	instruction modifies flag (either sets or resets depending on operands)
0	=	instruction resets flag
—	=	instruction's effect on flag is undefined
blank	=	instruction does not affect flag

80386 Programming Guide

Instruction	OF	SF	ZF	AF	PF	CF
AAA	—	—	—	TM	—	M
AAS	—	—	—	TM	—	M
AAD	—	M	M	—	M	—
AAM	—	M	M	—	M	—
DAA	—	M	M	TM	M	TM
DAS	—	M	M	TM	M	TM
ADC	M	M	M	M	M	TM
ADD	M	M	M	M	M	M
SBB	M	M	M	M	M	TM
SUB	M	M	M	M	M	M
CMP	M	M	M	M	M	M
CMPS	M	M	M	M	M	M
SCAS	M	M	M	M	M	M
NEG	M	M	M	M	M	M
DEC	M	M	M	M	M	
INC	M	M	M	M	M	
IMUL	M	—	—	—	—	M
MUL	M	—	—	—	—	M
RCL/RCR 1	M					TM
RCL/RCR count						TM
ROL/ROR 1	M					M
ROL/ROR count						M
SAL/SAR/SHL/SHR 1	M	M	M	—	M	M
SAL/SAR/SHL/SHR count		M	M	—	M	M
SHLD/SHRD		M	M	—	M	M
BSF/BSR		—	M	—	—	
BT/BTS/BTR/BTC		—		—	—	M
AND	0	M	M	—	M	0
OR	0	M	M	—	M	0
TEST	0	M	M	—	M	0
XOR	0	M	M	—	M	0

Appendix

D

Summary of 80386 Descriptors

Table D-1
Summary of 80386 Descriptors

Code	Type of Segment or Gate
0	<i>-reserved</i>
1	Available 286 TSS
2	LDT
3	Busy 286 TSS
4	Call Gate
5	Task Gate
6	286 Interrupt Gate
7	286 Trap Gate
8	<i>-reserved</i>
9	Available 386 TSS
A	<i>-reserved</i>
B	Busy 386 TSS
C	386 Call Gate
D	<i>-reserved</i>
E	386 Interrupt Gate
F	386 Trap Gate

80386 Programming Guide

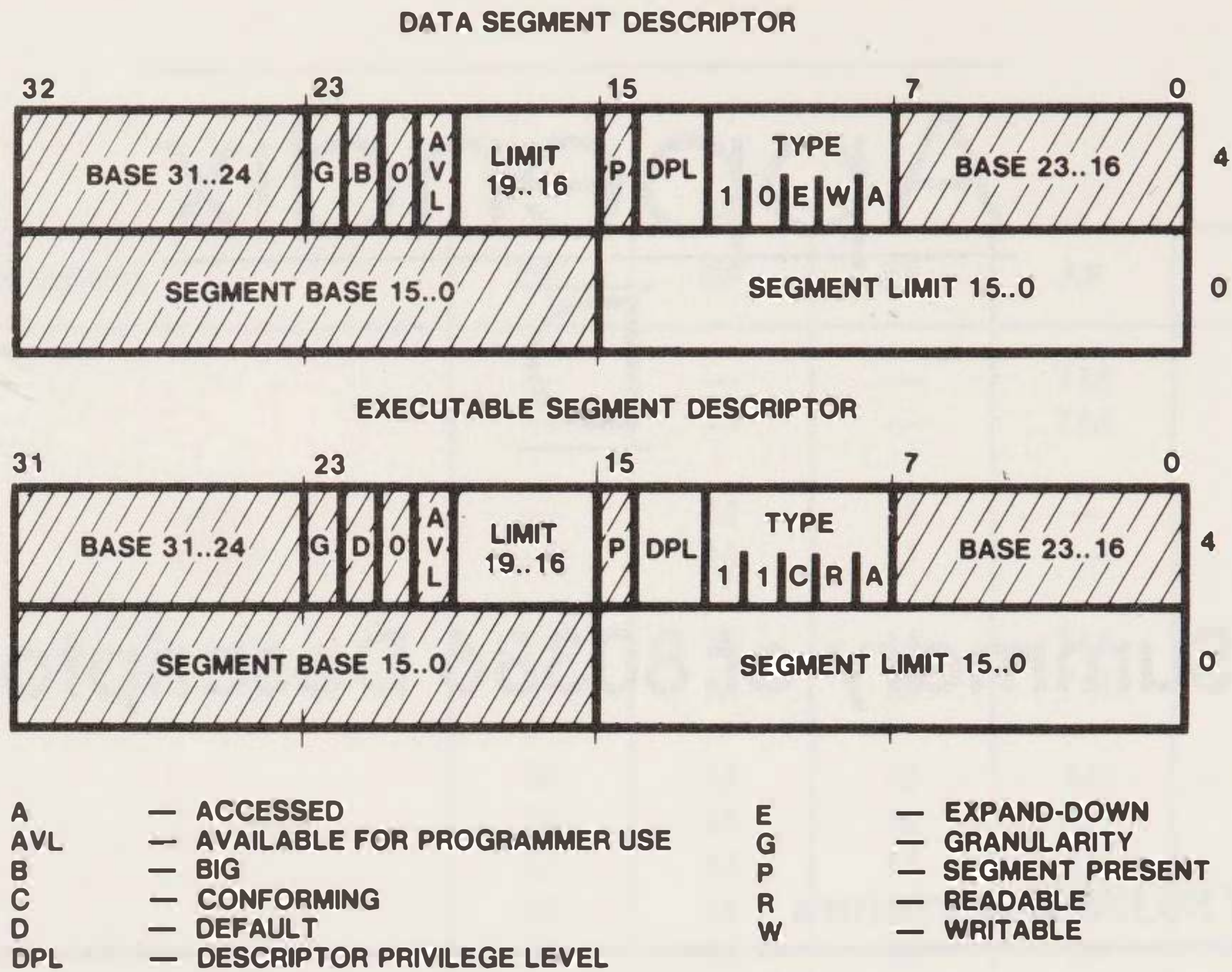


Figure D-1
Data and executable segment descriptors.

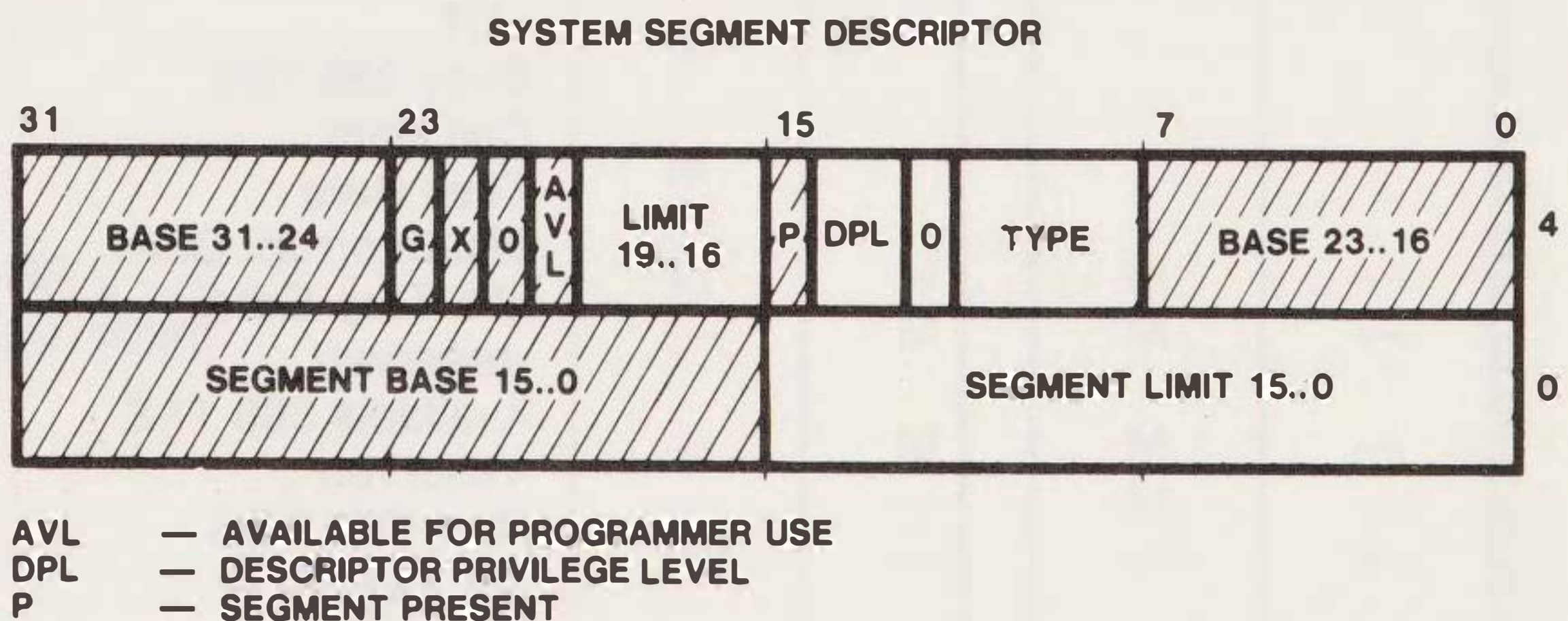


Figure D-2
System segment descriptor.

Appendix D

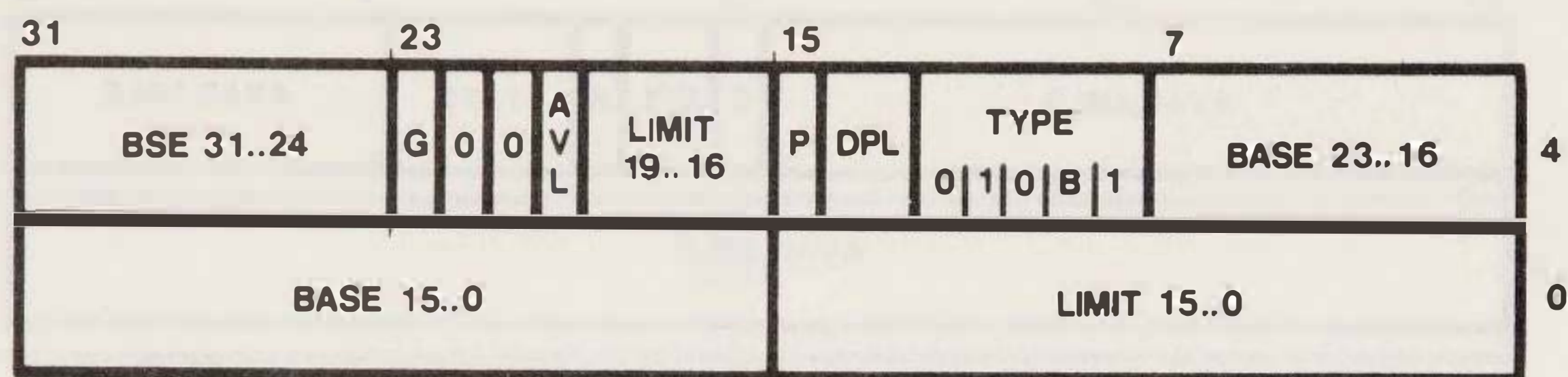


Figure D-3

Task state segment descriptor for 32-bit task state segment.

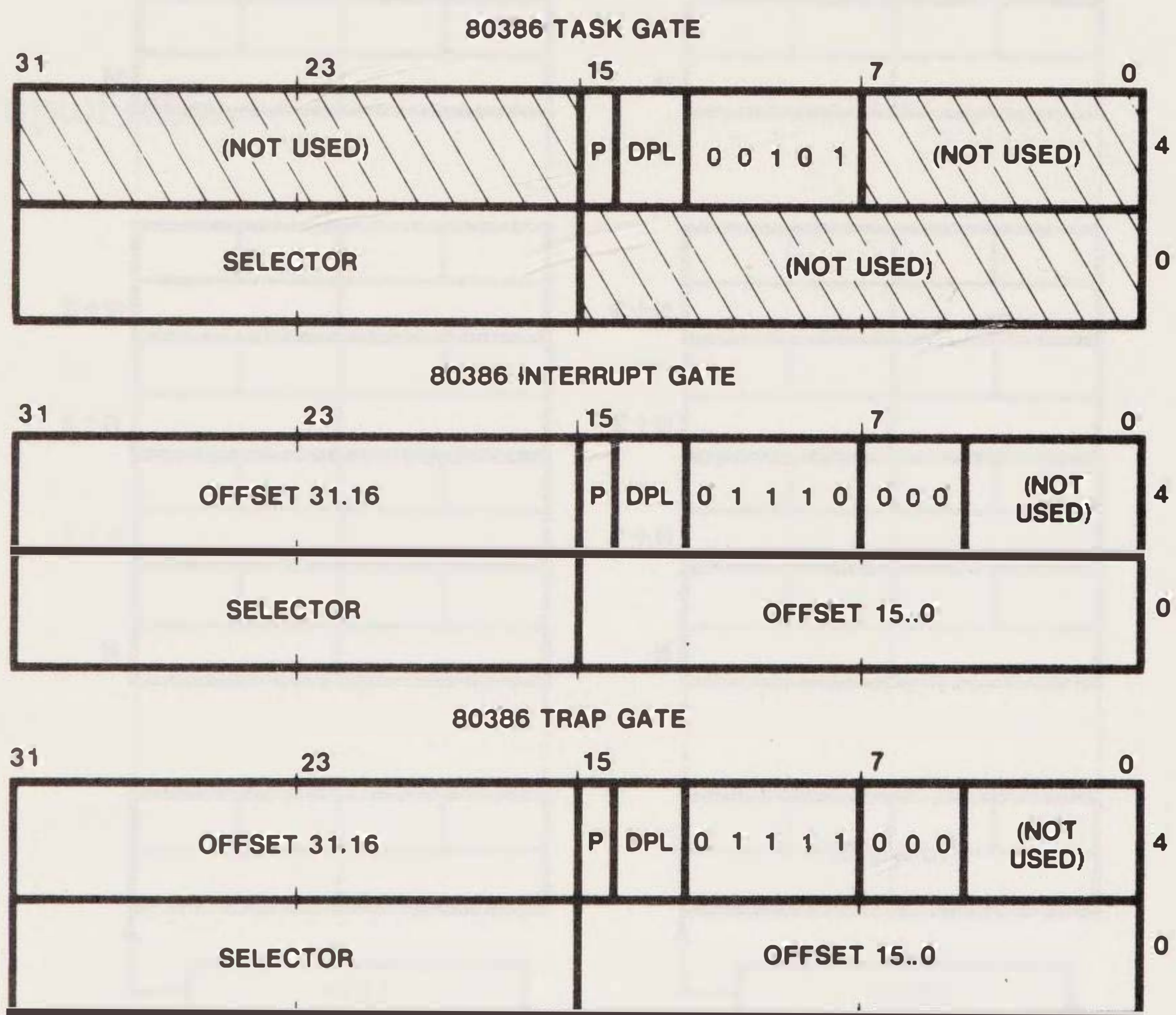


Figure D-4

Gate descriptors.



Figure D-5
Not-present descriptor.



Figure D-6
Global and local descriptor tables.

Appendix D

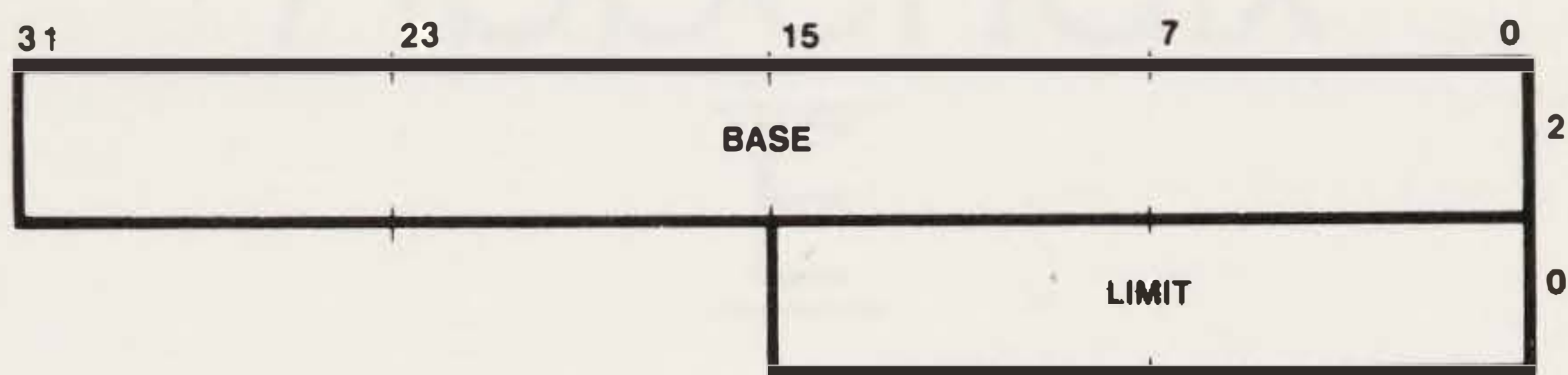


Figure D-7
Interrupt descriptor table.

Appendix

E

Differences Between 80286 and 80386 Processors

EXECUTING 80286 PROTECTED-MODE CODE

80286 CODE EXECUTES AS A SUBSET OF THE 80386

In general, programs designed for execution in protected mode on an 80286 execute without modification on the 80386, because the features of the 80286 are a subset of those of the 80386.

All the descriptors used by the 80286 are supported by the 80386 as long as the Intel-reserved word (last word) of the 80286 descriptor is zero.

The descriptors for data segments, executable segments, local descriptor tables, and task gates are common to both the 80286 and the 80386. Other 80286 descriptors — TSS segment, call gate, interrupt gate, and trap gate — are supported by the 80386. The 80386 also has new versions of descriptors for TSS segment, call gate, interrupt gate, and trap gate that support the 32-bit nature of the 80386. Both sets of descriptors can be used simultaneously in the same system.

For those descriptors that are common to both the 80286 and the 80386, the presence of zeros in the final word causes the 80386 to interpret these descriptors exactly as 80286 does; for example:

Base Address

The high-order eight bits of the 32-bit base address are zero, limiting base addresses to 24 bits.

Limit

The high-order four bits of the limit field are zero, restricting the value of the limit field to 64K.

Granularity bit

The granularity bit is zero, which implies that the value of the 16-bit limit is interpreted in units of one byte.

B-bit

In a data-segment descriptor, the B-bit is zero, implying that the segment is no larger than 64 K bytes.

D-bit

In an executable-segment descriptor, the D-bit is zero, implying that 16-bit addressing and operands are the default.

For formats of these descriptors and documentation of their use refer to the *iAPX 286 Programmer's Reference Manual*.

TWO WAYS TO EXECUTE 80286 TASKS

When porting 80286 programs to the 80386, there are two cases to consider:

1. Porting an entire 80286 system to the 80386, complete with 80286 operating system, loader, and system builder.

In this case, all tasks will have 80286 TSSs. The 80386 is being used as a faster 286.

2. Porting selected 80286 applications to run in an 80386 environment with an 80386 operating system, loader, and system builder.

In this case, the TSSs used to represent 80286 tasks should be changed to 80386 TSSs. It is theoretically possible to mix 80286 and 80386 TSSs, but the benefits are slight and the problems are great. It is recommended that all tasks in a 80386 software system have 80386 TSSs. It is not necessary to change the 80286 object modules themselves; TSSs are usually constructed by the operating system by the loader, or by the system builder. Refer to Chapter 16 for further discussion of the interface between 16-bit and 32-bit code.

DIFFERENCES FROM 80286

The few differences that do exist primarily affect operating system code.

WRAPAROUND OF 80286 24-BIT PHYSICAL ADDRESS SPACE

With the 80286, any base and offset combination that addresses beyond 16M bytes wraps around to the first megabyte of the 80286 address space. With the 80386, since it has a greater physical address space, any such address falls into the 17th megabyte. In the unlikely event that any software depends on this anomaly, the same effect can be simulated on the 80386 by using paging to map the first 64K bytes of the 17th megabyte of logical addresses to physical addresses in the first megabyte.

RESERVED WORD OF DESCRIPTOR

Because the 80386 uses the contents of the reserved word (last word) of every descriptor, 80286 programs that place values in this word may not execute correctly on the 80386.

NEW DESCRIPTOR TYPE CODES

Operating-system code that manages space in descriptor tables often uses an invalid value in the access-rights field of descriptor-table entries to identify unused entries. Access right values of 80H and 00H remain invalid for both the 80286 and 80386. Other values that were invalid on for the 80286 may be valid for the 80386 because of the additional descriptor types defined by the 80386.

RESTRICTED SEMANTICS OF LOCK

The 80286 processor implements the bus lock function differently than the 80386. Programs that use form of memory locking specific to the 80286 may not execute properly when transported to a specific application of the 80386.

The LOCK prefix and its corresponding output signal should only be used to prevent other bus masters from interrupting a data movement operation. LOCK may only be used with the following 80386 instructions when they modify memory. An undefined-opcode exception results from using LOCK before any other instruction.

- Bit test and change: BTS, BTR, BTC.
- Exchange: XCHG.
- One-operand arithmetic and logical: INC, DEC, NOT, and NEG.
- Two-operand arithmetic and logical: ADD, ADC, SUB, SBB, AND, OR, XOR.

A locked instruction is guaranteed to lock only the area of memory defined by the destination operand, but may lock a larger memory area. For example, typical 8086 and 80286 configurations lock the entire physical memory space. With the 80386, the defined area of memory is guaranteed to be locked against access by a processor executing a locked instruction on exactly the same memory area, i.e., an operand with identical starting address and identical length.

ADDITIONAL EXCEPTIONS

The 80386 defines new exceptions that can occur even in systems designed for the 80286.

Exception #6 — invalid opcode

This exception can result from improper use of the LOCK instruction.

Exception #14 — page fault

This exception may occur in an 80286 program if the operating system enables paging. Paging can be used in a system with 80286 tasks as long as all tasks use the same page directory. Because there is no place in an 80286 TSS to store the PDBR, switching to an 80286 task does not change the value of PDBR. Tasks ported from the 80286 should be given 80386 TSSs so they can take full advantage of paging.

DIFFERENCES FROM 80286 REAL-ADDRESS MODE

The few differences that exist between 80386 real-address mode and 80286 real-address mode are not likely to affect any existing 80286 programs except possibly the system initialization procedures.

BUS LOCK

The 80286 processor implements the bus lock function differently than the 80386. Programs that use forms of memory locking specific to the 80286 may not execute

properly if transported to a specific application of the 80386.

The LOCK prefix and its corresponding output signal should only be used to prevent other bus masters from interrupting a data movement operation. LOCK may only be used with the following 80386 instructions when they modify memory. An undefined-opcode exception results from using LOCK before any other instruction.

• Bit test and change: BTS, BTR, BTC.

• Exchange: XCHG.

• One-operand arithmetic and logical: INC, DEC, NOT, and NEG

• Two-operand arithmetic and logical: ADD, ADC, SUB, SBB, AND, OR, XOR.

A locked instruction is guaranteed to lock only the area of memory defined by the destination operand, but may lock a larger memory area. For example, typical 8086 and 80286 configurations lock the entire physical memory space. With the 80386, the defined area of memory is guaranteed to be locked against access by a processor executing a locked instruction on exactly the same memory area, i.e., an operand with identical starting address and identical length.

LOCATION OF FIRST INSTRUCTION

The starting location is OFFF0H (sixteen bytes from end of 32-bit address space) on the 80386 rather than OFFF0H (sixteen bytes from end of 24-bit address space) as on the 80286. Many 80286 ROM initialization programs will work correctly in this new environment. Others can be made to work correctly with external hardware that redefines the signals on A31-20.

INITIAL VALUES OF GENERAL REGISTERS

On the 80386, certain general registers may contain different values after RESET than on the 80286. This

80386 Programming Guide

should not cause compatibility problems, because the content of 8086 registers after RESET is undefined. If self-test is requested during the reset sequence and errors are detected in the 80386 unit, EAX will contain a nonzero value. ECS contains the component and revision identifier. Refer to Chapter 10 for more information.

MSW INITIALIZATION

The 80286 initializes the MSW register to FFF0H, but the 80386 initializes this register to 0000H. This difference should have no effect, because the bits that are different are undefined on the 80286. Programs that read the value of the MSW will behave differently on the 80386 only if they depend on the setting of the undefined, high-order bits.

DIFFERENCES FROM 80286 REAL-ADDRESS MODE

The 80286 processor implements the bus lock function differently than the 80386. This fact may or may not be apparent in 8086 programs, depending on how the V86 monitor handles the LOCK prefix. LOCKed instructions are sensitive to IOPL; therefore, software designers can choose to emulate its function. If, however, 8086 programs are allowed to execute LOCK directly, programs that use forms of memory locking specific to the 8086 may not execute properly when transported to a specific application of the 80386.

The LOCK prefix and its corresponding output signal should only be used to prevent other bus masters from interrupting a data movement operation. LOCK may only be used with the following 80386 instructions when they modify memory. An undefined-opcode exception results from using LOCK before any other instruction.

Bit test and change: BTS, BTR, BTC.

Exchange: XCHG.

One-operand arithmetic and logical: INC, DEC, NOT, and NEG.

Two-operand arithmetic and logical: ADD, ADC, SUBC, SBB, AND, OR, XOR.

A locked instruction is guaranteed to lock only the area of memory defined by the destination operand, but may lock a larger memory area. For example, typical 8086 and 80286 configurations lock the entire physical memory space. With the 80386, the defined area of memory is guaranteed to be locked against access by a processor executing a locked instruction on exactly the same memory area, i.e., an operand with identical starting address and identical length.

Appendix

F

Differences Between 8086 and 80386 Processor

REAL-ADDRESS MODE

DIFFERENCES FROM 8086

In general, the 80386 in real-address mode will correctly execute ROM-based software designed for the 8086, 8088, 80186, and 80188. Following is a list of the minor differences between 8086 execution on the 80386 and on an 8086.

1. Instruction clock counts.

The 80386 takes fewer clocks for most instructions than the 8086/8088. The areas most likely to be affected are:

Delays required by I/O devices between I/O operations.

Assumed delays with 8086/8088 operating in parallel with an 8087.

2. Divide Exceptions Point to the DIV instruction.

Divide exceptions on the 80386 always leave the saved CS:IP value pointing to the instruction that failed. On the 8086/8088, the CS:IP value points to the next instruction.

3. Undefined 8086/8088 opcodes.

Opcodes that were not defined for the 8086/8088 will cause exception 6 or will execute one of the new instructions defined for the 80386.

4. Value written by PUSH SP.

The 80386 pushes a different value on the stack for PUSH SP than the 8086/8088. The 80386 pushes the value of SP before SP is incremented as part of the push operation; the 8086/8088 pushes the value of SP after it is incremented. If the value pushed is important, replace PUSH SP instructions with the following three instructions:

```
PUSH BP
MOV BP, SP
XCHG BP, [BP]
```

This code functions as the 8086/8088 PUSH SP instruction on the 80386.

5. Shift or rotate by more than 31 bits.

The 80386 masks all shift and rotate counts to the low-order five bits. This MOD 32 operation limits the count to a maximum of 31 bits, thereby limiting

80386 Programming Guide

the time that interrupt response is delayed while the instruction is executing.

6. Redundant prefixes.

The 80386 sets a limit of 15 bytes on instruction length. The only way to violate this limit is by putting redundant prefixes before an instruction. Exception 13 occurs if the limit on instruction length is violated. The 8086/8088 has no instruction length limit.

7. Operand crossing offset 0 or 65,535.

On the 8086, an attempt to access a memory operand that crosses offset 65,535 (e.g., MOV a word to offset 65,535) or offset 0 (e.g. PUSH a word when SP = 1) causes the offset to wrap around modulo 65,536. The 80386 raises an exception in these cases — exception 13 if the segment is a data segment (i.e., if CS, DS, ES, FS, or GS is being used to address the segment), exception 12 if the segment is a stack segment (i.e., if SS is being used).

8. Sequential execution across offset 65,535.

On the 8086, if sequential execution of instructions proceeds past offset 65,535, the processor fetches the next instruction byte from offset 0 of the same segment. On the 80386, the processor raises exception 13 in such a case.

9. LOCK is restricted to certain instructions.

The LOCK prefix and its corresponding output signal should only be used to prevent other bus masters from interrupting a data movement operation. The 80386 always asserts the LOCK signal during an XCHG instruction with memory (even if the LOCK prefix is not used). LOCK may only be used with the following 80386 instructions when they update memory: BTS, BTR, BTC, XCHG, ADD, ADC, SUB, SBB, INC, DEC, AND, OR, XOR, NOT, and NEG. An undefined-opcode exception (interrupt 6) results from using LOCK before any other instruction.

10. Single-stepping external interrupt handlers.

The priority of the 80386 single-step exception is different from that of the 8086/8088. The change prevents an external interrupt handler from being single-stepped if the interrupt occurs while a program is being single-stepped. The 80386 single-step exception has higher priority than any external interrupt. The 80386 will still single-step through an interrupt handler invoked by the INT instructions or by an exception.

11. IDIV exceptions for quotients of 80H or 8000H.

The 80386 can generate the largest negative number as a quotient for the IDIV instruction. The 8086/8088 causes exception zero instead.

12. Flags in stack.

The setting of the flags stored by PUSHF, by interrupts, and by exceptions is different from that stored by the 8086 in bit positions 12 through 15. On the 8086 these bits are stored as ones, but in 80386 real-address mode bit 15 is always zero, and bits 14 through 12 reflect the last value loaded into them.

13. NMI interrupting NMI handlers.

After an NMI is recognized on the 80386, the NMI interrupt is masked until an IRET instruction is executed.

14. Coprocessor errors vector to interrupt 16.

Any 80386 system with a coprocessor must use interrupt vector 16 for the coprocessor error exception. If an 8086/8088 system uses another vector for the 8087 interrupt, both vectors should point to the coprocessor-error exception handler.

15. Numeric exception handlers should allow prefixes.

On the 80386, the value of CS:IP saved for coprocessor exceptions points at any prefixes before an ESC instruction. On 8086/8088 systems, the saved CS:IP points to the ESC instruction.

16. Coprocessor does not use interrupt controller.

The coprocessor error signal to the 80386 does not pass through an interrupt controller (an 8087 INT signal does). Some instructions in a coprocessor error handler may need to be deleted if they deal with the interrupt controller.

17. Six new interrupt vectors.

The 80386 adds six exceptions that arise only if the 8086 program has a hidden bug. It is recommended that exception handlers be added that treat these exceptions as invalid operations. This additional software does not significantly affect the existing 8086 software because the interrupts do not normally occur. These interrupt identifiers should not already have been used by the 8086 software, because they are in the range reserved by Intel. Table 14-2 describes the new 80386 exceptions.

18. One megabyte wraparound.

The 80386 does not wrap addresses at 1 megabyte in real-address mode. On members of the 8086 family, it is possible to specify addresses greater than one megabyte. For example, with a selector value 0FFFFH and an offset of 0FFFFH, the effective address would be 10FFEFH (1 Mbyte + 65519). The 8086, which can form addresses only up to 20 bits long, truncates the high-order bit, thereby “wrapping” this address to 0FFEFH. However, the 80386, which can form addresses up to 32 bits long does not truncate such an address.

80386 Programming Guide

VIRTUAL 8086 MODE

DIFFERENCES FROM 8086

In general, V86 mode will correctly execute software designed for the 8086, 8088, 80186, and 80188. Following is a list of the minor differences between 8086 execution on the 80386 and on an 8086.

1. Instruction clock counts.

The 80386 takes fewer clocks for most instructions than the 8086/8088. The areas most likely to be affected are:

Delays required by I/O devices between I/O operations.

Assumed delays with 8086/8088 operating in parallel with an 8087.

2. Divide exceptions point to the DIV instruction.

Divide exceptions on the 80386 always leave the saved CS:IP value pointing to the instruction that failed. On the 8086/8088, the CS:IP value points to the next instruction.

3. Undefined 8086/8088 opcodes.

Opcodes that were not defined for the 8086/8088 will cause exception 6 or will execute one of the new instructions defined for the 80386.

4. Value written by PUSH SP.

The 80386 pushes a different value on the stack for PUSH SP than the 8086/8088. The 80386 pushes the value of SP before SP is incremented as part of the push operation; the 8086/8088 pushes the value of SP after it is incremented. If the value pushed is important, replace PUSH SP instructions with the following three instructions:

```
PUSH BP
MOV BP, SP
XCHG BP, [BP]
```

This code functions as the 8086/8088 PUSH SP instruction on the 80386.

5. Shift or rotate by more than 31 bits.

The 80386 masks all shift and rotate counts to the low-order five bits. This MOD 32 operation limits the count to a maximum of 31 bits, thereby limiting the time that interrupt response is delayed while the instruction is executing.

6. Redundant prefixes.

The 80386 sets a limit of 15 bytes on instruction length. The only way to violate this limit is by putting redundant prefixes before an instruction. Exception 13 occurs if the limit on instructions length is violated. The 8086/8088 has no instruction length limit.

7. Operand crossing offset 0 or 65,535.

On the 8086, an attempt to access a memory operand that crosses offset 65,535 (e.g., MOV a word to offset 65,535) or offset 0 (e.g., PUSH a word when SP = 1) causes the offset to wrap around modulo 65,536. The 80386 raises an exception in these cases — exception 13 if the segment is a data segment (i.e., if CS, DS, ES, FS, or GS is being used to address the segment), exception 12 if the segment is a stack segment (i.e., if SS is being used).

8. Sequential execution across offset 65,535.

On the 8086, if sequential execution of instructions proceeds past offset 65,535, the processor fetches the next instruction byte from offset 0 of the same segment. On the 80386, the processor raises exception 13 in such a case.

9. LOCK is restricted to certain instructions.

The LOCK prefix and its corresponding output signal should only be used to prevent other bus masters from interrupting a data movement operation. The 80386 always asserts the LOCK signal during an XCHG instruction with memory (even if the LOCK prefix is not used). LOCK may only be used with the following 80386 instructions when they update memory: BTS, BTR, BTC, XCHG, ADD, ADC, SUB, SBB, INC, DEC, AND, OR, XOR, NOT, and NEG. An undefined-opcode exception (interrupt 6) results from using LOCK before any other instruction.

10. Single-stepping external interrupt handlers.

The priority of the 80386 single-step exception is different from that of the 8086/8088. The change prevents an external interrupt handler from being single-stepped if the interrupt occurs while a program is being single-stepped. The 80386 single-step exception has higher priority than any external interrupt. The 80386 will still single-step through an interrupt handler invoked by the INT instructions or by an exception.

11. IDIV exceptions for quotients of 80H or 8000H.

The 80386 can generate the largest negative number as a quotient for the IDIV instruction. The 8086/8088 causes exception zero instead.

12. Flags in stack.

The setting of the flags stored by PUSHF, by interrupts, and by exceptions is different from that stored by the 8086 in bit positions 12 through 15. On the 8086 these bits are stored as ones, but in V86 mode bit 15 is always zero, and bits 14 through 12 reflect the last value loaded into them.

13. NMI interrupting NMI handlers.

After an NMI is recognized on the 80386, the NMI interrupt is masked until an IRET instruction is executed.

14. Coprocessor errors vector to interrupt 16.

Any 80386 system with a coprocessor must use interrupt vector 16 for the coprocessor error exception. If an 8086/8088 system uses another vector for the 8087 interrupt, both vectors should point to the coprocessor-error exception handler.

15. Numeric exception handlers should allow prefixes.

On the 80386, the value of CS:IP saved for coprocessor exceptions points at any prefixes before an ESC instruction. On 8086/8088 systems, the saved CS:IP points to the ESC instruction itself.

16. Coprocessor does not use interrupt controller.

The coprocessor error signal to the 80386 does not pass through an interrupt controller (an 8087 INT signal does). Some instructions in a coprocessor error handler may need to be deleted if they deal with the interrupt controller.

Appendix

G

Overview of the 80287 and 80387 Numerical Data Processors

80287 and 80387 Numerics Coprocessors

The 80287 and 80387 are high-performance floating point coprocessors for 80386-based systems. The 80287 makes numerics power available to low-cost 80386 designs, while the 80387 provides enhanced functionality and the highest numerics performance available for 32-bit microprocessors. Both implement the IEEE 754 floating point standard, with high-precision 80-bit architectures and full support for single, double, and extended precision operations. Both coprocessors offer substantial performance enhancement over numerics software, are binary-compatible with the industry-standard 8087 numerics coprocessor, and both are fully supported by Intel and third-party high-level languages, as well as the Intel standard numerics libraries.

Product Highlights

80287 and 80387

- . High-performance 80-bit internal architectures
- . Implement IEEE 754 floating point standard
- . Datatypes include 32-bit single real, 64-bit double real, 80-bit extended real, 16-bit word integer, 32-bit short integer, 64-bit long integer, and 18-bit BCD integertypes
- . Object code compatible with 8087
- . Optimized interface with 80386 processor for highest possible floating point performance
- . Directly extends 80386 instruction set to include trigonometric, logarithmic, exponential, and arithmetic instructions for all datatypes
- . Operation completely conforms to 80386 native mode operation

80386 Programming Guide

80387 only	Support	$0 \leq x \leq \pi/4$	Cos, Simultaneous Sin-Cos, Unlimited Argument Range
Full 32-bit interface to 80386 local bus			
Enhanced trigonometric support			
Overall performance 1.8 million double-precision Whetstones/second			
CHMOS III technology			

Product Description

The 80287 and 80387 provide high-performance floating point capabilities for 80386 designs, with the 80287 being particularly well-suited for cost-sensitive applications and the 80387 for designs that require maximum performance. The 80387 is the latest entry in the Intel numerics coprocessor family, which started with the 8087 in 1979 and continued with the 80287 in 1982.

The 80387 incorporates the same philosophy followed throughout the family. First, implement the IEEE 754 floating point standard. This allows quick system design by providing a numerics solution that already implements a standard and is guaranteed correct. Second, remain object code compatible with previous members of the family — the 8087 and 80287. This allows all previous software developed for 86 family numerics applications to be available for 80386 designs. Finally, provide an enhancement in performance to keep floating point performance improvements in line with microprocessor performance improvements, allowing 80386-based products to be leaders in numerics performance as well as overall performance.

The following table summarizes the key differences between the 80287 and 80387:

	80287	80387
Process Technology	HMOS III	CHMOS III
Package	40-pin Cerdip	68-pin Ceramic Grid Array
Data Interface Width	16-bit	32-bit
Clock Speeds	5, 8, 10, 12 MHz	12, 16 MHz
Trigonometric	Tan, Arctan,	Tan, Arctan, Sin,

Programmer Model

Both the 80287 and 80387 contain a stack of eight 80-bit registers for numerics function computations. They also support seven datatypes: 32-bit short real, 64-bit long real, 80-bit extended real, 16-bit word integer, 32-bit short integer, 64-bit long integer, and 18-digit packed BCD integer. The 80287 and 80387 hold all numbers in the extended real format internally. Load instructions automatically convert operands represented in memory as 16-, 32-, or 64-bit integers, 32- or 64-bit floating point numbers, or 18-digit packed BCD numbers into extended real format. Store instructions automatically perform the reverse type conversion. This capability allows numerics applications to view data in the most appropriate form without concern for type conversions.

The 80287 and 80387 provide the full set of IEEE-compatible computational instructions. Additionally, commonly used constants are provided to again simplify development of numerics applications. The instruction sets provided are summarized in the following table.

80287 and 80387 Computational Instructions

Add Real	Square Root
Add Integer	Scale (fast multiply/divide by power of 2)
Subtract Real	Partial Remainder
Subtract Integer	Round to Integer
Multiply Real	Extract Exponent and Significand
Multiply Integer	Absolute Value
Divide Real	Change Sign
Divide Integer	Test for Zero
Compare Real	Examine Top of Stack
Compare Integer	

80287 and 80387 Constant Instructions

Load Zero	Load log ₂ 10
Load One	Load log ₂ e
Load Pi	Load log ₁₀ 2
	Load ln 2

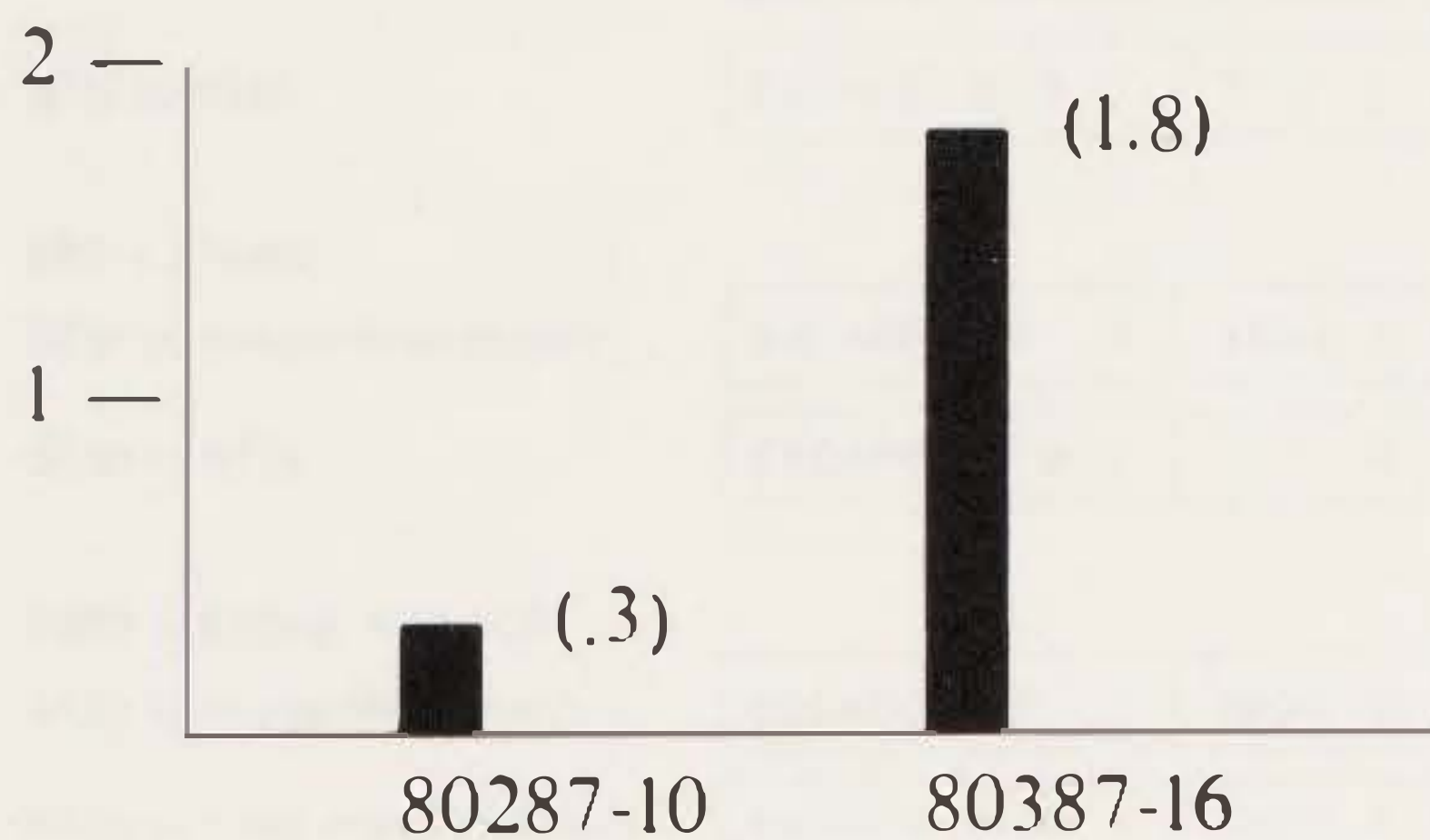
Transcendental Instructions

	80287	80387
Tangent	$0 \leq x \leq \pi/4$	full range
Arctangent	full range	full range
Sine	n.a.	full range
Cosine	n.a.	full range
Simultaneous Sine and Cosine	n.a.	full range
$2^x - 1$	$0 \leq x \leq .5$	$0 \leq x \leq .5$
$y \cdot \log_2 x$	full range	full range
$y \cdot \log_2 (x+1)$	full range	full range

The above set of instructions allow development of all types of numerics applications, such as solids modeling, mechanical simulation, robot control, and scientific analysis to name just a few. This is a powerful addition to the already powerful capabilities made available by the 80386 processor.

Performance

The 80287 implements numerics algorithms orders of magnitude faster than software implementations. The 80387 is even faster. The performance of these coprocessors executing the standard Whetstone benchmark is shown below:



This level of performance embodied in the 80287 and 80387 allows development of powerful, numerics-intensive systems. And since the 80287 and 80387 are standard, easy-to-use coprocessors, the development of these systems is very straightforward and fast.

Appendix

H

Instruction Set of the 80287 Numerical Data Processor

Data Transfer		Optional 8, 16 Bit Displacement	Clock Count Range			
			32 Bit Real	32 Bit Integer	64 Bit Real	16 Bit Integer
FLD = LOAD	MF =		00	01	10	11
Integer/Real Memory to ST(0)	ESCAPE MF 1 MOD 0 0 0 R/M	DISP	38-56	52-60	40-60	46-54
Long Integer Memory to ST(0)	ESCAPE 1 1 1 MOD 1 0 1 R/M	DISP	60-68			
Temporary Real Memory to ST(0)	ESCAPE 0 1 1 MOD 1 0 1 R/M	DISP	53-65			
BCD Memory to ST(0)	ESCAPE 1 1 1 MOD 1 0 0 R/M	DISP	290-310			
ST(i) to ST(0)	ESCAPE 0 0 1 1 1 0 0 0 ST(i)		17-22			
FST = STORE						
ST(0) to Integer/Real Memory	ESCAPE MF 1 MOD 0 1 0 R/M	DISP	84-90	82-92	96-104	80-90
ST(0) to ST(i)	ESCAPE 1 0 1 1 1 0 1 0 ST(i)		15-22			
FSTP = STORE AND POP						
ST(0) to Integer/Real Memory	ESCAPE MF 1 MOD 0 1 1 R/M	DISP	86-92	84-94	98-106	82-92
ST(0) to Long Integer Memory	ESCAPE 1 1 1 MOD 1 1 1 R/M	DISP	94-105			
ST(0) to Temporary Real Memory	ESCAPE 0 1 1 MOD 1 1 1 R/M	DISP	52-58			
ST(0) to BCD Memory	ESCAPE 1 1 1 MOD 1 1 0 R/M	DISP	520-540			
ST(0) to ST(i)	ESCAPE 1 0 1 1 1 0 1 1 ST(i)		17-24			
FXCH = Exchange ST(i) and ST(0)	ESCAPE 0 0 1 1 1 0 0 1 ST(i)		10-15			

80386 Programming Guide

Comparison

FCOM = Compare

Integer/Real Memory to ST(0)	ESCAPE MF 0 MOD 0 1 0 R/M	DISP	60-70	78-91	65-75	72-86
ST(i) to ST(0)	ESCAPE 0 0 0 1 1 0 1 0 ST(i)		40-50			

FCOMP = Compare and Pop

Integer/Real Memory to ST(0)	ESCAPE MF 0 MOD 0 1 1 R/M	DISP	63-73	80-93	67-77	74-88
ST(i) to ST(0)	ESCAPE 0 0 0 1 1 0 1 1 ST(i)		45-52			

FCOMPP = Compare ST(1) to ST(0) and Pop Twice

ESCAPE 1 1 0 1 1 0 1 1 0 0 1	45-55
------------------------------	-------

FTST = Test ST(0)

ESCAPE 0 0 1 1 1 1 0 0 1 0 0	38-48
------------------------------	-------

FXAM = Examine ST(0)

ESCAPE 0 0 1 1 1 1 0 0 1 0 1	12-23
------------------------------	-------

Constants

		Optional 8.16 Bit Displacement	Clock Count Range			
			32 Bit Real	32 Bit Integer	64 Bit Real	16 Bit Integer
MF =			00	01	10	11
FLDZ = LOAD + 0.0 into ST(0)	ESCAPE 0 0 1 1 1 1 0 1 1 1 0		11-17			
FLD1 = LOAD + 1.0 into ST(0)	ESCAPE 0 0 1 1 1 1 0 1 0 0 0		15-21			
FLDPI = LOAD π into ST(0)	ESCAPE 0 0 1 1 1 1 0 1 0 1 1		16-22			
FLDL2T = LOAD $\log_2 10$ into ST(0)	ESCAPE 0 0 1 1 1 1 0 1 0 0 1		16-22			
FLDL2E = LOAD $\log_2 e$ into ST(0)	ESCAPE 0 0 1 1 1 1 0 1 0 1 0		15-21			
FLDLG2 = LOAD $\log_{10} 2$ into ST(0)	ESCAPE 0 0 1 1 1 1 0 1 1 0 0		18-24			
FLDLN2 = LOAD $\log_e 2$ into ST(0)	ESCAPE 0 0 1 1 1 1 0 1 1 0 1		17-23			

Arithmetic

FADD = Addition

Integer/Real Memory with ST(0)	ESCAPE MF 0 MOD 0 0 0 R/M	DISP	90-120	108-143	95-125	102-137
ST(i) and ST(0)	ESCAPE d P 0 1 1 0 0 0 ST(i)		70-100 (Note 1)			

FSUB = Subtraction

Integer/Real Memory with ST(0)	ESCAPE MF 0 MOD 1 0 R R/M	DISP	90-120	108-143	95-125	102-137
ST(i) and ST(0)	ESCAPE d P 0 1 1 1 0 R R/M		70-100 (Note 1)			

FMUL = Multiplication

Integer/Real Memory with ST(0)	ESCAPE MF 0 MOD 0 0 1 R/M	DISP	110-125	130-144	112-168	124-138
ST(i) and ST(0)	ESCAPE d P 0 1 1 0 0 1 R/M		90-145 (Note 1)			

Appendix H

FDIV = Division
Integer/Real Memory with ST(0)

ESCAPE	MF	0	MOD	1	1	R	R/M
--------	----	---	-----	---	---	---	-----

DISP

 215-225 230-243 220-230 224-238

ST(i) and ST(0)

ESCAPE	d	P	0	1	1	1	1	R	R/M
--------	---	---	---	---	---	---	---	---	-----

 193-203 (Note 1)

FSORT = Square Root of ST(0)

ESCAPE	0	0	1	1	1	1	1	0	1	0
--------	---	---	---	---	---	---	---	---	---	---

 180-186

FSCALE = Scale ST(0) by ST(1)

ESCAPE	0	0	1	1	1	1	1	1	0	1
--------	---	---	---	---	---	---	---	---	---	---

 32-38

FPREM = Partial Remainder of ST(0) ÷ ST(1)

ESCAPE	0	0	1	1	1	1	1	0	0	0
--------	---	---	---	---	---	---	---	---	---	---

 15-190

FRNDINT = Round ST(0) to Integer

ESCAPE	0	0	1	1	1	1	1	1	0	0
--------	---	---	---	---	---	---	---	---	---	---

 16-50

NOTE:

1. If P=1 then add 5 clocks.

Optional
8,16 Bit
Displacement

Clock Count Range

FXTRACT = Extract Components of St(0)

ESCAPE	0	0	1	1	1	1	0	1	0	0
--------	---	---	---	---	---	---	---	---	---	---

 27-55

FABS = Absolute Value of ST(0)

ESCAPE	0	0	1	1	1	0	0	0	0	1
--------	---	---	---	---	---	---	---	---	---	---

 10-17

FCHS = Change Sign of ST(0)

ESCAPE	0	0	1	1	1	0	0	0	0	0
--------	---	---	---	---	---	---	---	---	---	---

 10-17

Transcendental

FPTAN = Partial Tangent of ST(0)

ESCAPE	0	0	1	1	1	1	0	0	1	0
--------	---	---	---	---	---	---	---	---	---	---

 30-540

FPATAN = Partial Arctangent of ST(0) ÷ ST(1)

ESCAPE	0	0	1	1	1	1	0	0	1	1
--------	---	---	---	---	---	---	---	---	---	---

 250-800

F2XM1 = $2^{ST(0)} - 1$

ESCAPE	0	0	1	1	1	1	0	0	0	0
--------	---	---	---	---	---	---	---	---	---	---

 310-630

FYL2X = ST(1) • Log₂ |ST(0)|

ESCAPE	0	0	1	1	1	1	0	0	0	1
--------	---	---	---	---	---	---	---	---	---	---

 900-1100

FYL2XP1 = ST(1) • Log₂ |ST(0) + 1|

ESCAPE	0	0	1	1	1	1	1	0	0	1
--------	---	---	---	---	---	---	---	---	---	---

 700-1000

Processor Control

FINIT = Initialize NPX

ESCAPE	0	1	1	1	1	0	0	0	1	1
--------	---	---	---	---	---	---	---	---	---	---

 2-8

FSETPM = Enter Protected Mode

ESCAPE	0	1	1	1	1	0	0	1	0	0
--------	---	---	---	---	---	---	---	---	---	---

 2-8

FSTSW AX = Store Control Word

ESCAPE	1	1	1	1	1	0	0	0	0	0
--------	---	---	---	---	---	---	---	---	---	---

 10-16

FLDCW = Load Control Word

ESCAPE	0	0	1	MOD	1	0	1	R/M
--------	---	---	---	-----	---	---	---	-----

DISP

 7-14

FSTCW = Store Control Word

ESCAPE	0	0	1	MOD	1	1	1	R/M
--------	---	---	---	-----	---	---	---	-----

DISP

 12-18

FSTSW = Store Status Word

ESCAPE	1	0	1	MOD	1	1	1	R/M
--------	---	---	---	-----	---	---	---	-----

DISP

 12-18

80386 Programming Guide

FCLEX = Clear Exceptions	ESCAPE 0 1 1 1 1 1 0 0 0 1 0	2-8
FSTENV = Store Environment	ESCAPE 0 0 1 MOD 1 1 0 R/M DISP	40-50
FLDENV = Load Environment	ESCAPE 0 0 1 MOD 1 0 0 R/M DISP	35-45
FSAVE = Save State	ESCAPE 1 0 1 MOD 1 1 0 R/M DISP	205-215
FRSTOR = Restore State	ESCAPE 1 0 1 MOD 1 0 0 R/M DISP	205-215
FINCSTP = Increment Stack Pointer	ESCAPE 0 0 1 1 1 1 1 0 1 1 1	6-12
FDECSTP = Decrement Stack Pointer	ESCAPE 0 0 1 1 1 1 1 0 1 1 0	6-12
Clock Count Range		
FFREE = Free ST(i)	ESCAPE 1 0 1 1 1 0 0 0 ST(i)	9-16
FNOP = No Operation	ESCAPE 0 0 1 1 1 0 1 0 0 0 0	10-16

NOTES:

1. If mod = 00 then DISP = 0*, disp-low and disp-high are absent.
 If mod = 01 then DISP = disp-low sign-extended to 16-bits, disp-high is absent.
 If mod = 10 then DISP = disp-high; disp-low.
 If mod = 11 then r/m is treated as an ST(i) field.
2. If r/m = 000 then EA = (BX) + (SI) + DISP
 If r/m = 001 then EA = (BX) + (DI) + DISP
 If r/m = 010 then EA = (BP) + (SI) + DISP
 If r/m = 011 then EA = (BP) + (DI) + DISP
 If r/m = 100 then EA = (SI) + DISP
 If r/m = 101 then EA = (DI) + DISP
 If r/m = 110 then EA = (BP) + DISP
 If r/m = 111 then EA = (BX) + DISP

*Except if mod = 000 and r/m = 110 then EA = disp-high; disp-low

3. MF = Memory Format
 - 00-32-bit Real
 - 01-32-bit Integer
 - 10-64-bit Real
 - 11-16-bit Integer
4. ST(0) = Current stack top
ST(i) ;th register below stack top
5. d = Destination
 - 0-Destination is ST(0)
 - 1-Destination is ST(i)
6. P = Pop
 - 0-No pop
 - 1-Pop ST(0)
7. R = Reverse: When d= 1 reverse the sense of R
 - 0-Destination (op) Source
 - 1-Source (op) Destination
8. For FSQRT: $-0 \leq ST(0) \leq +\infty$
 For FXCALE: $-2^{15} \leq ST(1) < +2^{15}$ and
 ST(1) integer
 For F2XM1: $0 \leq ST(0) \leq 2^{-1}$
 For FYL2X: $0 < ST(0) < \infty$
 $-\infty < ST(1) < \infty$
 For FYL2XP1: $0 \leq |ST(0)| < (2 - \sqrt{2})$
 $-\infty < ST(1) < \infty$
 For FPTAN: $0 \leq ST(0) \leq \pi/4$
 For FPATAN: $0 \leq ST(0) < ST(1) < +\infty$
9. ESCAPE bit pattern is 11011.

Appendix

I

Instruction Set of the 80387 Numerical Data Processor

80387 EXTENSIONS TO THE 80386 INSTRUCTION SET

Instructions for the 80387 assume one of the five forms shown in the following table. In all cases, instructions are at least two bytes long and begin with the bit pattern 11011B, which identifies the ESCAPE class of instruction. Instructions that refer to memory operands specify addresses using the 80386 addressing modes.

OP=Instruction opcode, possible split into two fields OPA and OPB

MF=Memory Format

00—32-bit real

01—32-bit integer

10—64-bit real

11—16-bit integer

P =Pop

0—Do not pop stack

1—Pop stack after operation

ESC=11011

d =Destination

0—Destination is ST(0)

1—Destination is ST(i)

R XOR d = 0—Destination (op) Source

R XOR d = 1—Source (op) Destination

ST(i) = Register stack element i

000 = Slack top

001 = Second stack element

•

•

•

111 = Eighth stack element

		Instruction							Optional Fields	
		First Byte			Second Byte					
1	11011	OPA	1	MOD	1	OPB	R/M	SIB	DISP	
2	11011	MF	OPA	MOD		OPB	R/M	SIB	DISP	
3	11011	d	P	OPA	1	1	OPB	ST(i)		
4	11011	0	0	1	1	1	1	OP		
5	11011	0	1	1	1	1	1	OP		
		15-11	10	9	8	7	6	5	4	3 2 1 0

80386 Programming Guide

Instruction	Encoding			Clock Count Range			
	Byte 0	Byte 1	Optional Bytes 2-6	32-Bit Real	32-Bit Integer	64-Bit Real	16-Bit Integer
DATA TRANSFER							
FLD = Load^a							
Integer/real memory to ST(0)	ESCMF 1	MOD 000 R/M	SIB/DISP	20	45-52	25	61-65
Long integer memory to ST(0)	ESC 111	MOD 101 R/M	SIB/DISP		56-67		
Extended real memory to ST(0)	ESC 011	MOD 101 R/M	SIB/DISP		44		
BCD memory to ST(0)	ESC 111	MOD 100 R/M	SIB/DISP		266-275		
ST(i) to ST(0)	ESC 001	11000 ST(i)			14		
FST = Store							
ST(0) to integer/real memory	ESC MF 1	MOD 010 R/M	SIB/DISP	44	79-93	45	82-95
ST(0) to ST(i)	ESC 101	11010 ST(i)			11		
FSTP = Store and Pop							
ST(0) to integer/real memory	ESCMF 1	MOD 011 R/M	SIB/DISP	44	79-93	45	82-95
ST(0) to long integer memory	ESC 111	MOD 111 R/M	SIB/DISP		80-97		
ST(0) to extended real	ESC 011	MOD 111 R/M	SIB/DISP		53		
ST(0) to BCD memory	ESC 111	MOD 110 R/M	SIB/DISP		512-534		
ST(0) to ST(i)	ESC 101	11001 ST(i)			12		
FXCH = Exchange							
ST(i) and ST(0)	ESC 001	11001 ST(i)			18		
COMPARISON							
FCOM = Compare							
Integer/real memory to ST(0)	ESC MF 0	MOD 010 R/M	SIB/DISP	26	56-83	31	71-75
ST(i) to ST(0)	ESC 000	11010 ST(i)			24		
FCOMP = Compare and pop							
Integer/real memory to ST	ESC MF 0	MOD 011 R/M	SIB/DISP	26	56-63	31	71-75
ST(i) to ST(0)	ESC 000	11011 ST(i)			26		
FCOMPP = Compare and pop twice							
ST(1) to ST(0)	ESC 110	1101 1001			26		
FTST = Test ST(0)							
	ESC 001	1110 0100			28		
FUCOM = Unordered compare							
	ESC 101	11100 ST(i)			24		
FUCOMP = Unordered compare and pop							
	ESC 101	11101 ST(i)			26		
FUCOMPP = Unordered compare and pop twice							
	ESC 010	1110 1001			26		
FXAM = Examine ST(0)							
	ESC 001	11100101			30-38		
CONSTANTS							
FLDZ = Load +0.0 into ST(0)							
	ESC 001	1110 1110			20		
FLD1 = Load +1.0 into ST(0)							
	ESC 001	1110 1000			24		
FLDPI = Load pi into ST(0)							
	ESC 001	1110 1011			40		
FLDL2T = Load log₂(10) into ST(0)							
	ESC 001	1110 1001			40		

Shaded areas indicate instructions not available in 8087/80287.

NOTE:

a. When loading single- or double-precision zero from memory, add 5 clocks.

Appendix I

Instruction	Encoding			Clock Count Range			
	Byte 0	Byte 1	Optional Bytes 2-6	32-Bit Real	32-Bit Integer	64-Bit Real	16-Bit Integer
CONSTANTS (Continued)							
FLDL2E = Load $\log_2(e)$ into ST(0)	ESC 001	1110 1010				40	
FLDLG2 = Load $\log_{10}(2)$ into ST(0)	ESC 001	1110 1100				41	
FLDLN2 = Load $\log_e(2)$ into ST(0)	ESC 001	1110 1101				41	
ARITHMETIC							
FADD = Add							
Integer/real memory with ST(0)	ESC MF 0	MOD 000 R/M	SIB/DISP	24-32	57-72	29-37	71-85
ST(i) and ST(0)	ESC d P 0	11000 ST(i)				23-31 ^b	
FSUB = Subtract							
Integer/real memory with ST(0)	ESC MF 0	MOD 10 R R/M	SIB/DISP	24-32	57-82	28-36	71-83 ^c
ST(i) and ST(0)	ESC d P 0	1110 R R/M				26-34 ^d	
FMUL = Multiply							
Integer/real memory with ST(0)	ESC MF 0	MOD 001 R/M	SIB/DISP	27-35	61-82	32-57	76-87
ST(i) and ST(0)	ESC d P 0	1100 1 R/M				29-57 ^e	
FDIV = Divide							
Integer/real memory with ST(0)	ESC MF 0	MOD 11 R R/M	SIB/DISP	89	120-127 ^f	94	136-140 ^g
ST(i) and ST(0)	ESC d P 0	1111 R R/M				88 ^h	
FSQRT ⁱ = Square root	ESC 001	1111 1010				122-129	
FSCALE = Scale ST(0) by ST(1)	ESC 001	1111 1101				67-86	
FPREM = Partial remainder	ESC 001	1111 1000				74-155	
FPREM1 = Partial remainder (IEEE)	ESC 001	1111 0101				95-186	
FRNDINT = Round ST(0) to integer	ESC 001	1111 1100				66-80	
FXTRACT = Extract components of ST(0)	ESC 001	1111 0100				70-76	
FABS = Absolute value of ST(0)	ESC 001	1110 0001				22	
FCHS = Change sign of ST(0)	ESC 001	1110 0000				24-25	

NOTES:

- Add 3 clocks to the range when $d = 1$.
- Add 1 clock to **each** range when $R = 1$.
- Add 3 clocks to the range when $d = 0$.
- typical = 52 (When $d = 0$, 46-54, typical = 49).
- Add 1 clock to the range when $R = 1$.
- 135-141 when $R = 1$.
- Add 3 clocks to the range when $d = 1$.
- $-0 \leq ST(0) \leq +\infty$.

80386 Programming Guide

Instruction	Encoding			Clock Count Range
	Byte 0	Byte 1	Optional Bytes 2-6	
TRANSCENDENTAL				
FCOS^k = Cosine of ST(0)	ESC 001	1111 1111		123-772
FPTAN^k = Partial tangent of ST(0)	ESC 001	1111 0010		191-4971
FPATAN = Partial arctangent	ESC 001	1111 0011		314-487
FSIN^k = Sine of ST(0)	ESC 001	1111 1110		122-7711
FSINCOS^k = Sine and cosine of ST(0)	ESC 001	1111 1011		194-6021
F2XM1^l = 2 ^{ST(0)} - 1	ESC 001	1111 0000		211-476
FYL2X^m = ST(1) * log ₂ (ST(0))	ESC 001	1111 0001		120-538
FYL2XP1ⁿ = ST(1) * log ₂ (ST(0) + 1.0)	ESC 001	1111 1001		257-547
PROCESSOR CONTROL				
FINIT = Initialize NPX	ESC 011	1110 0011		33
FSTSW AX = Store status word	ESC 111	1110 0000		13
FLDCW = Load control word	ESC 001	MOD 101 R/M	SIB/DISP	19
FSTCW = Store control word	ESC 101	MOD 111 R/M	SIB/DISP	15
FSTSW = Store status word	ESC 101	MOD 111 R/M	SIB/DISP	15
FCLEX = Clear exceptions	ESC 011	1110 0010		11
FSTENV = Store environment	ESC 001	MOD 110 R/M	SIB/DISP	103-104
FLDENV = Load environment	ESC 001	MOD 100 R/M	SIB/DISP	71
FSAVE = Save state	ESC 101	MOD 110 R/M	SIB/DISP	375-376
FRSTOR = Restore state	ESC 101	MOD 100 R/M	SIB/DISP	308
FINCSTP = Increment stack pointer	ESC 001	1111 0111		21
FDECSTP = Decrement stack pointer	ESC 001	1111 0110		22
FFREE = Free ST(i)	ESC 101	1100 0 ST(i)		18
FNOP = No operations	ESC 001	1101 0000		12

NOTES:

j. These timings hold for operands in the range $|x| < \pi/4$. For operands not in this range, up to 76 additional clocks may be needed to reduce the operand.

k. $0 \leq |ST(0)| < 2^{63}$.

l. $-1.0 \leq ST(0) \leq 1.0$.

m. $0 \leq ST(0) < \infty$, $-\infty < ST(1) < +\infty$.

n. $0 \leq |ST(0)| < (2 - \text{SQRT}(2))/2$, $-\infty < ST(1) < +\infty$.

Appendix I

MOD (Mode field) and R/M (Register/Memory specifier) have the same interpretation as the corresponding fields of 80386 instructions (refer to *80386 Programmer's Reference Manual*)

SIB (Scale Index Base) byte and DISP (displacement) are optionally present in instructions that have MOD and R/M fields. Their presence depends on the values of MOD and R/M, as for 80386 instructions.

The instruction summaries that follow assume that the instruction has been prefetched, decoded, and is ready for execution; that bus cycles do not require wait states; that there are no local bus HOLD request delaying processor access to the bus; and that no exceptions are detected during the instruction execution. If the instruction has mOD and R/M fields that call for both base and index registers, add one clock.

Acknowledgements

Grateful acknowledgment is due for the use of the following materials in this book:

Figure 6-10 is from Intel Corporation's 386 SYSTEM BUILDER USER'S GUIDE FOR XENIX 286 SYSTEMS, Order No. 122299-002, Copyright 1986, Intel Corporation, Santa Clara, CA. Used with permission.

Figures 2-1, 2-2, 2-3, 2-4, 2-5, 2-6, 2-7, 2-8 are from Intel Corporation's 80386 DATA SHEET, Order No. 231630-49, Copyright 1986, Intel Corporation, Santa Clara, CA. Used with permission.

Figures 1-3, 1-4, 8-1, 8-2, 8-3, 8-4, 8-7, 8-8, 8-9, 8-10, 8-11, 8-12, 8-13, 8-14, 8-15, 8-16, 8-17, 8-18, 8-19, 8-20, 8-21; Tables 8-1, 8-2, 8-3, 8-5, 8-6, 8-7, and 8-8 are from Intel Corporation's 80386 HARDWARE REFERENCE MANUAL, Order No. 231732-001, Copyright 1986, Intel Corporation, Santa Clara, CA. Used with permission.

Figure 2-9 and Table 2-1 are from Intel Corporation's INTRODUCTION TO THE 80386, Order No. 231746-001, Copyright 1986, Intel Corporation, Santa Clara, CA. Used with permission.

Figures 4-13, 4-14, 4-15, 4-16, 4-17, 4-18, 4-19, and 4-20 are from Intel Corporation's MICROSYSTEM COMPONENTS HANDBOOK, VOLUME I: MICROPROCESSORS, Order No. 230843-003, Copyright 1986, Intel Corporation, Santa Clara, CA. Used with permission.

Figures 4-4, 4-5, 4-6, 4-7, 4-8, and 4-9 are from Intel Corporation's MICROSYSTEM COMPONENTS HANDBOOK, VOLUME II: PERIPHERALS, Order No. 230843-003, Copyright 1986, Intel Corporation, Santa Clara, CA. Used with permission.

Figures 4-12, 5-10, 5-11, 5-12, 5-13, 5-14, 5-2, 5-3, 5-4, 5-5, 5-6, 5-7, 5-8, 5-9, 6-1, 6-4, 6-5, 6-6, 6-7, 6-8, 7-1, 7-2, 7-3, 7-4, 7-5, 7-6, 8-5, 8-6, D-1, D-2, D-3, D-4, D-5, D-6, D-7; Tables 4-3, 5-1, 5-2, 7-1, 7-2, 7-3, 7-4, 7-5, 7-6, 7-7, D-1; Appendices A, B, C, E, and F are from Intel Corporation's 80386 PROGRAMMER'S REFERENCE MANUAL, Order No. 230985-001, Copyright 1986, Intel Corporation, Santa Clara, CA. Used with permission.

Figures 1-13 and 1-14 are from Intel Corporation's 80386 SLIDE NOTEBOOK, Copyright 1986, Intel Corporation, Santa Clara, CA. Used with permission. Figures 6-2, 6-3, and 6-9 are from Intel Corporation's 80386 SYSTEM SOFTWARE WRITER'S

GUIDE, Order No. 231499-001, Copyright 1987, Intel Corporation, Santa Clara, CA. Used with permission.

Appendix G is from Intel Corporation's INTEL'S 386 FAMILY (October 1985 version — Order No. 231635-001). Used with permission.

Appendix H is from the 80287 specification (entitled 80287 80-BIT HMOS-NUMERIC PROCESSOR EXTENSION — Intel document 200920-005, dated February 1986). Used with permission.

Appendix I consists of pp. 33-36 of the 80387 specification (entitled 80387 80-BIT CHMOS III NUMERIC PROCESSOR EXTENSION — Intel document 231920-002, dated January 1987). Used with permission.

Index

' (around characters), 34
[,] (around memory addresses), 34
: (in front of labels), 34
? (undefined initial value), 81
; (indicating a comment), 34

A

A (accessed) bit, 162
Aborts, 213, 214
Absolute value, 114-115
Accessed (A) bit in segment descriptor, 162
Accumulator, 34
Active-low, 232
ADC instruction, 59
Adding entries to a list, 110, 114
Add-on 80386 CPU boards, 238
Addressable domain restrictions, 175, 178
Addressing in subroutines, 116-117
Addressing modes, 45-48
 definitions, 45
 examples, 46
 execution time, 48
 list, 45
 overlapped execution, 5, 48
 summary, 47
Address pipelining, 231, 247
Address size, 78-79
Address size override, 78
Address status (ADS#) signal, 238, 245
Address translation, 8
 8086, 157-158
 paging, 167-173
 protected mode, 165-166
 segmentation, 157-167
ADS# signal, 238, 245
AF (auxiliary carry flag), 38
Aliases (of descriptors), 202
ALIGN directive, 117
Aligning memory accesses, 117
Alphabetical listing of instruction set, 49-53
Applications, 9-20
 list, 9-10
Arithmetic, multiple-precision, 108-109
Arithmetic and logical instructions, 58-59, 66
 carry, including of, 59
 LEA, 58
 order of operands, 58
ARPL instruction, 178
Array bounds, 102-103, 119
Array manipulation, 101-103
Artificial intelligence, 18
ASCII, 42
ASCII (unpacked BCD) arithmetic instructions, 66
ASCII characters, 104-108
Asserted state, 232
Associative caches, 261, 262-263, 265
Autodecrementing, 63, 102, 105
Autoincrementing, 63, 102, 105
Auxiliary Carry flag, 38

B

- B (big) bit in data segment descriptor, 177
- B (busy) bit in task state descriptor, 195
 - use, 201
- Background tasks, 204
- Back link (in task state segment), 192, 201
 - removing, 201
 - unused, 202
- Barrel shifter, 5, 79
- Base, 46
- Based index addressing with displacement, 45
 - examples, 46
 - extra clock cycle, 48
- Base pointer, 35, 116
- Base register, 34
- Baud rate generator, 129
- BCD (binary-coded-decimal) representation, 42
- Benign exceptions, 220
- Big (B) bit in data segment descriptor, 177
- BIOS, 61
- Bit manipulation, 91-92
 - bit manipulation instructions, 91
 - DEC, 92
 - examples, 91-92
 - INC, 92
 - logical instructions, 91-92
- Bit manipulation instructions, 66, 70, 88, 91
- Bit pattern comparison, 92
- Bit scan instructions, 66, 80
- Bit string, 42
- Bit testing, 96-97
- BLD386 program, 183-184, 205-207
- Block fill, 113-114
- Block input/output, 122, 125
- Block size (in a cache), 260, 261
- Boolean values, 71
- BOUND instruction, 102-103, 119, 213, 220
- Bounds check exception, 220
- Bounds checking, 102-103
- Brackets around memory addresses, 34, 90
- Breakpoints, 220, 226
- BS16# signal, 238
- BT instruction, 91, 97, 136
- BTC instruction, 91
- BTR instruction, 91, 136
- BTS instruction, 91
- Buffered I/O, 136, 141-142
- Buffer pointers, 142
- Bus-based systems, 235-236
- Bus control logic, 255-257
- Bus cycle types, 242, 247
- Bus interface unit, 8
- Bus master, 242
- Bus operation, 242-249
 - cycle types, 242, 247
 - interrupt acknowledge cycles, 247-249
 - non-pipelined cycles, 242-247
 - pipelined cycles, 247
- Bus performance, 249
- Busy (B) bit in task state descriptor, 195, 201
- BUSY# signal, 241
- Busy task, 201
- Byte, 42
- Byte-addressable registers, 36
- Byte enable (BE) signals, 235
- BYTE PTR operator, 56

C

- C (conforming) bit, 183
- Cache, 31, 154, 260-268
 - associative, 261, 262-264, 265
 - block size, 261
 - considerations, 260
 - controller, 260-261
 - direct-mapped, 261, 264-265
 - fully associative, 261, 262-263
 - non-cacheable memory, 267-268
 - organizations, 261-262
 - performance, 268
 - set-associative, 262, 265
 - types, 261-262
 - updating, 265-267
- Cache coherency, 267
- Cache controller, 260-261, 265, 266, 268
- Cache organization, 261-262

- Cache performance, 268
- Cache updating, 265-267
- CAD/CAM/CAE systems, 14-17
 - application areas, 17
 - tasks, 14-17
 - typical systems, 14
- Call gates, 179-183
 - entry points, 180
 - format, 179-180
 - multiple gates, 181
 - new privilege level, 180
 - parameter count, 181
 - stack change, 181
- Carry flag, 38
 - arithmetic, 59
 - bit manipulation instructions, 38, 66
 - bit test, 97
 - CMP, 99
 - comparing equal values, 118
 - DEC, 79
 - INC, 79
 - logical instructions, 38, 79
 - multiple-precision arithmetic, 79, 108
 - purpose, 38
 - shift instructions, 80, 94
- Case statement, 104
- CF (carry flag), 38
- Character, 42
- Character manipulation, 104-107
- CL register, use in shifts, 59
- Clear instruction, 59
- Clearing bits, 91
- Clearing registers, 59
- CLI instruction, 140
- CLK2 signal, 242
- Clock counts for instructions, 272-285
- CMP (compare) instruction, 59, 97-100
- Code conversion, 107-108, 113
- Code prefetch unit, 8
- Code segment (CS) register, 38
 - interrupt descriptor table, 139
- Coherency, 267
- Common programming errors, 118-119
- Compaq Deskpro 386, 10
- Comparing bit patterns, 92
- Comparing values, 97-100
 - order of operands, 97
 - signed values, 100
 - unsigned values, 99-100
- Compatibility, 29-30, 36, 37, 305-313
- Compilation, 5
- Complementing bits, 91, 92
- Conditional jumps, 60-61, 96-100
 - comparisons, 60, 97-100
 - frequently used, 55, 60-61
 - JECXZ, 71
 - LOOP, 71, 100
 - mnemonics, 60
 - signed, 100
 - summary, 72-73
 - unsigned, 99-100
- Conditional LOOPS, 71, 101
- Conditional REPs, 63, 107, 113, 114
- Conforming code segments, 183
- Context of a task, 188
- Contributory exceptions, 220-222
- Control registers, 40
 - page directory base register (CR3), 169
- Control transfer restrictions, 178-183
- Conversion instructions, 61, 62
 - flags, effects on (none), 80
- Coprocessors, 250-255, 315-328
 - addresses, 250
 - bus cycles, 253
 - characteristics, 250
 - 80287/80387 recognition, 254
 - exceptions, 255
 - instructions, 250
 - instruction sets, 319-328
 - interfaces, 250-253
 - performance, 315
 - recognition, 254
 - signals, 241
 - task switch, 200, 255
- Coprocessor signals, 241
- Copy (MOV) instruction, 54
- Count (ECX) register, 34, 59, 71, 101
- CPL (current privilege level), 178, 183

D

- D (default operand/address size) bit, 75, 163
- D (dirty) bit, 172
- DAA instruction, 66, 109
- DAS instruction, 66
- Data breakpoints, 226, 228
- Data bus sizing, 231, 234, 235
- Data/Control (D/C#) signal, 236
- Data manipulation instructions, 58-59, 63-70
- Data register, 34
- Data segment descriptor, 300
- Data segment registers, 38, 40
- Data size, 56, 75
- Data storage, order of bytes, 44
- Data structures, 109-111
- Data transfer instructions, 54-58, 61-62
 - flags, effect on (none), 79
 - frequently used, 54-58
 - general, 61-62
- Data types, 42-45
 - names, 42
 - storage, 44-45
- DB directive, 81, 84
- DD directive, 81, 84
- Deadlock, 27
- Debug exceptions, 219, 229
- Debugging features, 226-227
- Debug registers, 226, 227-229
 - address registers, 226, 228
 - control register, 228
 - status register, 228
- Decimal arithmetic instructions, 66, 109
- DEC instruction, 79, 90, 92
 - bit manipulation, 92
- Decisionmaking, 96-101
- Default address/operand size bit, 75, 163
- Decoding, 8
- Demand-paged system, 23
- Departmental computing, 11
- Descriptor privilege level (DPL), 162, 178, 183
 - task gate, 197-199
 - task state segment descriptors, 195

- Descriptors, 161-167
 - creation, 183-184
 - formats, 299-302
 - gate, 179-180
 - summary, 299-303
 - tables, 302-303
 - task state segment, 195-196
 - types, 299
- Descriptor tables, 163-164
- Destination index (DI) register, 35
- DF (direction flag), 38, 63
- Direct addressing, 45, 46
- Direction flag (DF), 38, 63
- Directives (assembler), 81, 84
- Direct-mapped caches, 261, 264-265
- Dirty (D) bit in page table entries, 172
 - setting, 173
- Disabling interrupts, 140
- Displacement, 45, 46, 90
- Divide errors, 219, 220
- DMA signals, 242
- DMA transfers, 122
- Domain restrictions, 178
- Double faults, 220-222
- Double-length shifts, 88, 92, 94, 96
 - program example, 96
- Double word, 42
- DPL (descriptor privilege level), 162, 178, 183, 195
- DQ directive, 81
- DRAM, 259
- DRAM controller, 258, 259
- DT directive, 81
- Dump utility, 45
- DUP operator, 84
- Dword, 42
- DWORD override, 101
- DWORD PTR operator, 56, 90
- DW directive, 81, 84
- Dynamic data bus sizing, 231, 235
- Dynamic RAM, 259
- Dynamic RAM controller, 258, 259

E

- E (expansion-direction) bit, 176-178
- Early-out multiplication algorithm, 79, 88
- Effective addresses, 45, 47, 56
- EFLAGS register, 37-38.
 - See also* flags.
- 8-bit registers, 36-37
- 8086/8088 comparison, 29-30, 80-81, 83-84
 - 80286 differences, 80-81, 83
 - memory capacity, 30
 - real mode, 309-311
 - register restrictions, 37
 - speed, 30
 - virtual 8086 mode, 312-313
- 80286 comparison, 29-30, 80-81, 83
 - memory capacity, 30
 - new instructions, 83
 - porting programs, 306
 - protected mode, 305-308
 - real mode, 308
 - register restrictions, 37
 - speed, 30
- 80287 numeric coprocessor, 250, 315-317
 - instruction set, 319-323
 - interface, 250-251
 - recognition, 254
- 80387 numeric coprocessor, 241, 315-317
 - instruction set, 325-328
 - interface, 253
 - read cycles, 253
 - recognition, 254
- 8237 DMA controller, 149
- 8250 ACE, 127-129
- 8253 PIT, 132-133
- 8254 PIT, 132-133
- 8255 PPI, 127, 130-132
- 8259 Programmable Interrupt Controller (PIC), 143-147
 - EOI command, 144, 147
 - features, 143-144
 - initialization sequence, 144
- 82258 Advanced Direct Memory Access Coprocessor (ADMA), 150
- 82384 Clock Generator, 239
- 82385 Cache Controller, 261, 265, 268
- EM (emulate coprocessor) bit, 254
- Emulating coprocessor instructions, 254, 255
- Enabling interrupts, 140
- END directive, 81
- ENTER instruction, 116
- Entry point restrictions, 175, 178-183
- EQU directive, 81, 84
- ERROR# signal, 241, 254
 - coprocessor recognition, 254
 - exceptions, 255
- Errors, programming, 118-119
- ESC instruction, 71, 250, 253, 255
- ET (extension type) bit, 239, 254
- EVEN directive, 117
- Even parity, 38
- Exception error codes, 217-219
- Exception handlers, 183
 - conforming segments, 183
 - gates, 217-218
 - privilege level, 217
 - task implementations, 199
- Exceptions, 137, 211-226
 - classes, 213-215
 - conditions, 219-220
 - coprocessor, 255
 - new 80386 features, 212
 - sources, 212-214
 - types, 212-213
- Exchanging elements, 102
- Executable segment descriptor, 300
- Execution unit, 8
- Expand-down segments, 176-178
- Expansion-direction (E) bit, 176-178
- Extended registers, 34-36
- External signals, 232-242

F

- Faster programs, 117-118
- Faults, 213-214
- File server, 13
- Filling a pipeline, 8
- Filling memory, 113-114
- Flag cross-reference, 295-296
- Flags, 37-38, 295-298
 - compatibility, 37
 - cross-reference, 295-296
 - diagram, 37
 - functions, 297
 - instructions, effects of, 79-80
 - major flags, 37-38
 - summary, 297-298
- FLAGS register, 37
- Flag summary, 297-298
- Floating point (IEEE 754) numbers, 42-43, 250
- Flushing page cache, 173
- Foreground tasks, 204
- Frame, 22, 116
- Frame pointer, 116
- Framing, 129
- Frequently used instructions, 48, 54-62
 - arithmetic and logical instructions, 58-59
 - data transfer instructions, 54, 56-58
 - program control instructions, 60-61
- Fully associative caches, 261, 262-263
- Functional units, 7-8
- Future advances, 30-31

G

- G (granularity) bit, 163
- Gate descriptors, 179-180, 301
 - not-present bit, 223
- GB, 2, 3
- Generalized architecture, 5
- Gigabit, 42
- Gigabyte, 2, 3
- Global descriptor table, 163, 302
 - shared memory, 202

- Global descriptor table register, 163
- GOTOs, 9
- Granularity (G) bit, 163
- Graphics, 14-15

H

- Handshaking signals, 132
- Head pointer, 142
- Hex digit conversion, 113
- High-level languages, 29
 - features required, 29
 - instructions, 77
- Hit (in a cache), 260
- Hit ratio, 260
- HLDA signal, 242
- HOLD signal, 242

I

- IDT register, 139-140
- IEEE 754 (floating point) representation, 42-43, 250
- IF (interrupt enable flag), 38, 137
- Image processing, 19-20
- Immediate addressing, 45, 46
- INC instruction, 90, 98
 - bit manipulation, 92
- Increment with carry, 79
- Index, 45
- Index addressing, 45, 46
- Indirect jumps, 60, 104
- IN (input) instruction, 56-57, 124-125, 236
- Initial state of processor after RESET, 239-241
- INS instruction, 105, 125
- Instruction breakpoints, 226-229
- Instruction continuation, 214
- Instruction decode unit, 8
- Instruction execution unit, 8
- Instruction fetch, 8
- Instruction pointer, 36, 37
 - interrupt descriptor table, 139

- Instruction prefetch queue, 8, 185
 - Instruction restart, 23, 214
 - Instruction resumption, 23
 - Instruction set, 48-74, 272-293
 - address length, 75, 79
 - alphabetical listing, 49-53
 - arithmetic and logical instructions, 58-59, 63, 66-67
 - clock count summary, 272-285
 - data manipulation instructions, 58-59, 63, 66-67
 - data transfer instructions, 54, 56-58, 61-62
 - encoding, 286-293
 - frequently used, 48, 54-61
 - general, 61-74
 - listing, 49-53
 - operand length, 75, 79
 - other, 71
 - program control instructions, 60-61, 71, 72-74
 - restrictions, 175, 184
 - Integers, 42
 - Interleaved memory, 259
 - Interrupt acknowledge cycles, 236, 242, 247-249
 - Interrupt controller, 143-147
 - Interrupt descriptor table, 139-140, 214-217, 303
 - default values, 139-140
 - Interrupt descriptor table (IDT) register, 139-140
 - Interrupt-driven I/O, 122
 - Interrupt enable flag (IF), 38, 137
 - Interrupt gates, 216-217
 - Interrupt latency, 172
 - Interrupt priority, 143, 214
 - Interrupt-related instructions, 140
 - Interrupts, 137-147
 - inputs, 137
 - priority, 214
 - response, 60-61, 138-139
 - Interrupt service routine examples, 140-143
 - Interrupt tasks, 199
 - Interrupt type, 137, 8-15
 - Interrupt vectors, 139, 143, 267
 - INT (software interrupt) instruction, 60-61
 - NTO instruction, 219, 220
 - INTR input, 137, 242
 - INT 3 instruction, 220
 - Invalid operation code exception, 220
 - Invalid task state segment faults, 223
 - Inverting bits, 91, 92
 - I/O addresses, 56, 122-124
 - capacity, 56
 - instructions, 124-125
 - isolated, 122-124
 - memory-mapped, 122, 124, 236
 - non-segmented, 56, 122
 - reserved, 57, 124
 - I/O address register (DX), 124, 125
 - I/O devices, definition of, 236
 - I/O examples, 133, 136
 - I/O guard map, 204
 - I/O instructions, 124-125, 236
 - I/O methods, 121-122
 - I/O permission bit maps, 203-205
 - characteristics, 205
 - example, 205
 - uses, 203-204
 - IOPL, 203
 - V86 mode, 203
 - I/O privilege levels, 203
 - IRET instruction, 140, 141, 217
 - Isolated input/output, 56, 122-124
- ## J
- JECXZ instruction, 71, 101
 - JMP instruction, 60
 - Jumps, 9, 60
 - elimination, 117
 - indirection, 60
 - Jump table, 104
- ## K
- Keyboard decoding, 104
 - Keyboard operations on IBM PC, 61

L

- Large memory model, 159
- LAR instruction, 166, 225
- LDT (local descriptor table) register, 163
- LEA instruction, 58
 - arithmetic uses, 58
- Least recently used (LRU) algorithm, 23
- LEAVE instruction, 117
- LIDT instruction, 140
- Limit checking, 102-103, 175
- Linear address, 161, 167
 - division for paging, 168
- Linked lists, 109-110
- List manipulation, 109-110, 114
- Local descriptor table, 163, 302
 - shared memory, 202
- Local descriptor table register (LDTR), 163
- LOCK prefix, 71, 307, 308, 310
- LOCK# signal, 237-238
- Logical addresses, 8
- Logical instructions, 91-92
 - Carry, effect on, 38, 79
- Long shifts, 5
- Lookup tables, 61, 103-104, 118
- Looping, 101
- LOOP instruction, 71, 101
- Low byte first storage, 44
- LRU (least recently used) algorithm, 23
- LSL instruction, 166, 225
- LSS instruction, 226
- LTR instruction, 196-197, 208

M

- Machine language, 5
- Machine state, 25
- Macro Assembler notation, 34
- Mailbox, 141
- Maximum value, 111-112
- MB, 3

- Megabyte, 3
- Memory banks, 232, 259
- Memory boards, 3
- Memory capacity, 2-3
 - comparison to 80286, 2
 - comparison to 8086, 2
 - expansion, 3
 - Intel processors, 2, 4
 - virtual, 30
- Memory chips, 3
- Memory interface, 255-259
- Memory/IO (M/IO#) signal, 236
- Memory management systems, 153-186
 - initialization, 184-185
- Memory management unit (MMU), 21, 154-155
 - 80386, 21
 - on-chip vs. separate, 155
- Memory-mapped I/O, 122, 124, 236
 - non-cacheable, 267-268
- Memory models, 159
- Memory-to-memory operations, 56, 66, 90
- Memory transfer control signals, 232-239
- Memory units, 3
- Misaligned transfers, 235, 236
- Miss, 260
- MMU, 21, 154-155
- Mnemonics, alternative, 60
- Most significant bit, 38
- Motorola 68000 family, 104, 155, 214
- MOV (move) instructions, 54, 56, 89, 90
 - addressing modes, 56
 - debug registers, 227-228
 - exchange, 61
 - order of operands, 54
- MOVS instruction, 63, 66, 106
- MP (math present) bit, 255
- MS-DOS entry points, 61
- MS-DOS software, 3-4
 - 80286 protected mode, 4, 167
- Multibyte array elements, 46, 48, 102, 103
- Multiple-bit shifts, 79
- Multiple-precision arithmetic, 108-109
- Multiplication methods, 79

Multiplication using LEA, 58
Multitasking, 4, 25-27, 188-190

- advantages, 26-27
- applications, 4-5
- arbitration, 27
- background, 204
- controller, 26
- disadvantages, 27
- foreground, 204
- functions, 25
- I/O, 203-205
- I/O-bound, 26, 27
- personal computer, 189
- priority, 25

Multiuser systems, 11, 13, 27-28

- administration, 28
- I/O, 204
- problems, 28
- uses, 27-28

Multiword shifts, 94, 96

N

NA# signal, 238, 247
Negated state, 232
Nested task (NT) flag, 190, 199, 201
Next address request (NA#) signal, 238, 247
NMI input, 137, 242
Non-busy task, 201
Non-cacheable memory, 24, 267-268
Nonexistent address, 247
Nonmaskable interrupt, 137, 242

- vector, 220, 242

Non-overlapping segments, 159-160
Non-pipelined bus cycles, 242-247
Notation, macroassembler, 34
NOT instruction, 79
Not-present descriptor, 302
NT (nested task) flag, 190, 199, 201
Null selector, 165, 166
Numeric coprocessor, 250-255.
See also coprocessors.

O

OF (overflow flag), 37
OFFSET operator, 84
Offsets, 42, 157, 168
Operand size, 75, 79, 89
Operand size override, 75, 79
Operating modes, 155-156
Operating system support, 29
OS restrictions, avoiding of, 180-181
OS/2, 4, 30, 80
OUT instruction, 56-57, 124-125, 236
Output service routines, 141-142
OUTS instruction, 106, 125
Overlays, 21
Overflow, 100
Overflow flag (OF), 37
Overrides:

- address/operand, 75, 79
- segment, 158-159

Overrun, 129

P

P (segment present) bit, 162
Packed BCD, 42

- instructions, 66

Page, 22
Page cache, 173-174
Page directory, 167-168
Page directory base register (CR3), 169
Page fault, 21, 225-226

- error code, 218
- example, 22-23
- invalid stack pointer, 226
- meaning, 225
- multiple faults, 23
- task switches, 225

Page frame, 22, 170
Page present (P) bit, 172
Page protection, 175-176

- Page size, 24
 - Page table, 22-23, 167-172
 - elements, 171
 - format, 171-172
 - size, 171
 - starting address, 172
 - Page unit, 8
 - Paging, 22-25, 167-174
 - after segmentation, 167
 - demand-paged system, 23
 - disadvantages, 24
 - 80386 support, 154
 - example, 22-23
 - page size, 24
 - translation, 167-171
 - working set, 23
 - Paging (PG) bit, 167, 185
 - Paging unit, 8
 - Parameter count in task gates, 181
 - Parameter passing, 115-117
 - Parity flag (PF), 38
 - Parsing command lines, 107
 - Pattern match, 113
 - PCs, 11-14
 - PE (protection enable) bit, 155, 184, 185
 - PEREQ signal, 241, 250
 - Performance comparisons, 30
 - Personal computers, 11-14
 - Personal System/2 computers (IBM), 1, 2, 16
 - PF (parity flag), 38
 - PG (paging) bit, 167, 185
 - Physical addresses, 8
 - Pipelined bus cycles, 247
 - Pipelining, 6-9
 - example, 8
 - operation, 8
 - programming techniques, 8-9
 - Plant controller, 26-27
 - PL/M function interface, 115
 - Pointer, 42
 - Polling, 122
 - POP instruction, 57-58
 - POPAD instruction, 57
 - Power users, 11
 - Precharge time, 259
 - Prefetch unit, 8
 - Privileged instructions, 184
 - Privilege levels, 175-176
 - changing, 181, 182
 - conforming code segments, 183
 - descriptors, 175
 - exception handlers, 217
 - pages, 175-176
 - user/supervisor systems, 175-176
 - V86 mode, 191
 - Process, 188
 - Processor control instructions, 71, 78
 - Processor-detected exceptions, 212-214
 - Program control instructions, 60-61, 71, 72-74
 - frequently used, 60-61
 - general, 71, 72-74
 - Program counter, 36, 37
 - Programmable I/O chips, 125-133
 - advantages, 125-126
 - disadvantages, 126-127
 - Programmed exceptions, 213
 - Programmed I/O, 121-122
 - Programming errors, common, 118-119
 - Protection, 28-29, 174-184
 - Protection enable (PE) bit, 155, 184, 185
 - Protection exceptions, 176, 224-225
 - Protection model instructions, 71, 78
 - Pseudo-operations, 81, 84
 - PS/2 computers, 1, 2, 16
 - PTR (pointer) operator, 56, 84, 90
 - PUSH instruction, 57-58
 - PUSHAD instruction, 58
- ## Q
- Quad word, 42
 - storage example, 45
 - ? (undefined initial value), 81, 84
 - Queues, 8
 - Qword, 42

R

RCL instruction, 93, 94
 RCR instruction, 93, 94
 Readable (R) bit, 176
 Read-modify-write sequence, 238
 Read/write (R/W) bit, 172
 READY# signal, 238-239, 246
 Real mode, 6, 155-156
 differences from 8086, 309-311
 differences from 80286, 308
 Real-time systems, 24, 204
 Reduced-instruction-set (RISC) machines, 31, 48
 Reentrancy (of tasks), 195
 Refresh, 259
 Register addressing, 45, 46
 Register indirect addressing, 45, 46
 Registers, 34-40
 byte-addressable, 36-37
 clearing, 59
 control, 40
 debug, 40, 227-228
 8-bit, 36-37
 flags, 37-38
 general-purpose, 34-37
 limitations (8086/80286), 37
 limitations (80386), 48
 segment, 38, 39, 40
 setting flags from, 79, 98
 16-bit, 36
 specialized, 40
 special uses, 88
 system address, 39, 40
 test, 40
 user, 34-38
 word-addressable, 36
 Removing entries from a list, 114
 REP (repeat) prefix, 63, 106
 conditional versions, 63, 107, 112-114
 examples, 66, 107, 112-114
 I/O, 125
 limitations, 63
 order of steps, 63
 parsing command lines, 107

Reserved interrupts, 137-138
 RESET signal, 239-241
 coprocessor, 254
 initial state, 239, 240
 startup address, 239
 Restart, 23
 Resume flag (RF), 227
 RET instruction to change privilege levels, 182
 RF (resume flag), 227
 RISC machines, 31, 48
 Robotics, 17-18
 ROL instruction, 93, 94
 ROR instruction, 94
 RPL (requestor's privilege level), 178
 RS-232 interface, 127

S

SAL instruction, 94, 95
 SAR instruction, 94, 95
 SBB instruction, 59
 Scaled index in addressing, 46
 Scaled indexing, 102, 103, 104
 Scientific users, 11
 Scientific workstations, 11
 Segmentation methods, 157-167
 8086, 157-161
 protected mode, 161-167
 Segmentation unit, 8
 Segment descriptor, 161-163
 Segment limit, 161
 Segment not present exceptions, 223-224
 Segment overrides, 158-159
 not allowed, 159
 Segment present (P) bit, 162
 Segment registers, 38-40, 157
 default assignments, 158
 80386 additions, 40
 hidden parts, 166
 length, 38
 overrides, 158-159
 validating, 224
 Segments, 5, 21

- characteristics, 157
- size limit, 30, 154
- virtual memory, 21
- Segment type, 161
- Segment unit, 8
- Selector, 161, 164-167
 - examples, 164-165
 - format, 164
 - null, 165
- Self-modifying code, 176
- Semaphores, 238
- Sequential operation, 8
- Set associative caches, 262, 265
- SETcc instruction, 71, 113
- Setting bits, 91-92
- SF (sign flag), 38
- Shared facilities, 28
- Shared memory, 202, 238
- Shift counts, 59
- Shift instructions, 59, 79, 92-96
- SHL instruction, 94, 95
- SHLD instruction, 94, 96
- SHR instruction, 94, 95
- SHRD instruction, 94, 96
- Signal pin summary, 233-234
- Signal processing, 19-20
 - tasks, 19-20
 - typical applications, 20
- Signed conditional jumps, 100
- Sign-extended conversions, 61, 62, 115
- Sign extension, 94
- Sign flag (SF), 38
- 16-bit registers, 36
- Slow memory, interfacing of, 238-239
- Small memory model, 159
- Snooping, 268
- Software interrupt (INT instruction), 60-61, 213, 220
- Software state, 192
- Source index (SI) register, 36
- Speeding up programs, 117-118
- Stack-based parameter passing, 116-117
- Stack cache, 31
- Stack exceptions, 224
- Stack pointer, 35
 - limitation on use, 46
 - subroutines, 116-117
- Stacks (software), 110-111
- Stack segment (SS) register, 38, 224, 226
- Stale data, 265
- Standard buses, 235-236
- Startup address, 239
- Startup signals, 239-241
- Step size in string instructions, 66
- STI instruction, 140
- Stop bit, 127
- STR instruction, 197
- String, 42
- String comparison, 113
- String instructions, 63, 66, 68
 - examples, 66, 105-107
 - flag effects, 80
 - list, 66, 105-106
 - REP prefix, 63, 106-107
 - step size, 63
 - step direction, 63
- String length, 112-113
- String manipulation, 63, 66, 68, 104-107
- String primitives, 63, 105-106
- Subdepartmental computing, 11, 13
- SUB instruction, 58-59, 79
- Supervisor level, 172, 175-176
- Suspending a task, 189
- Swapping, 23-24
 - example, 22-23
 - problems, 24
- Switching modes, 6
- System address and segment registers, 39
- System Builder (BLD386) program, 183-184, 205-208
- System segment descriptor, 300
- System-wide resources, 200

T

Table lookup, 61, 103-104, 118

Tags, 173, 260, 261
 length in caches, 265

Tail pointer, 142

Task address spaces, 202

Task context, 188

Task gate descriptors, 197-199

Task gates, 197-199

Tasking, 25-27, 188-190
 advantages, 190
 80386 features, 190-199
 examples, 188-189
 functionalization, 27
 initialization, 205-208
 interrupt service routines, 199
 priority, 189
 reentrancy, 195
 requirements, 25
 V86 mode, 191, 208

Task isolation, 27, 190

Task linking, 201-202

Task management, 187-209

Task priority, 189

Task register, 196-197

Task state, 188, 199, 200

Task state segment (TSS) descriptors, 195-196,
 301
 initialization, 205-208

Task state segments, 192-195
 dummy, 207-208
 dynamic information, 192
 exceptions, 222, 223
 extended version, 192, 193, 195
 initialization, 205-208
 minimum, 192
 reading, 195
 software state, 192
 static information, 192
 validation, 224
 V86 version, 208
 writing, 195

Task switch, 25, 26, 199-200

 advantages, 200
 coprocessor, 200, 255
 disadvantages, 200
 instructions causing, 200
 privilege levels, 200
 procedure, 199-200

Terabyte (TB), 3

Testing bits, 91, 97

Testing memory, 98

TEST instruction, 59, 90, 97, 98
 setting flags from register value, 79, 98

TF (trap flag), 38, 217, 227

Thrashing, 23

TI (table indicator) bit in selector, 164

Timing diagrams, 242-249

TLB, 173-174

Translation lookaside buffer (TLB), 173-174
 entry format, 173
 flushing, 173
 memory range, 173

Transparent routines, 140

Trap bit (of a task state segment), 227

Trap flag (TF), 38, 217, 227

Trap gates, 215, 216

Traps, 213-214

TS (task switched) bit, 200

Two's complement overflow, 100

Type checking, 175

Type conversions, 61

Type definitions, 75

U

UART, 126, 127-129

Unbuffered I/O, 133, 136, 140-141

Unix operating system, 17, 188, 192, 195

Unmarked numbers, 46, 90

Unpacked BCD, 42
 instructions, 66

Unsigned conditional jumps, 60, 99-100

USE directive, 75

User level (page privilege), 175-176

User segments, 175-176

User/supervisor (U/S) bit, 172
User/supervisor systems, 175-176, 203
Utility program, 183

V

Validating a TSS, 224
V86 mode, 6, 153, 156
 invalid exit, 225
 I/O permission maps, 204
 privilege level, 191
 tasking, 191
 task initialization, 208
Virtual devices, 203
Virtual 8086 (V86) mode, 6, 153, 156
 advantages, 6
 differences from 8086, 312-313
 IOPL, 203
Virtual 8086 monitor, 6, 154, 191, 203
Virtual machine, 6
Virtual memory, 5, 21-25
 advantages, 21
 capacity, 30
 implementations, 21-22, 162
 paged, 22-25
 segmented, 21
Virtual Mode (VM) flag, 156, 191
Visible part of a segment register, 166
VM (virtual mode) flag, 156, 191

W

Wait states, 239, 242, 249
Watchdog timer, 247
Weitek floating point chip set, 250
What-if analysis, 16-17
Word, 42
 storage example, 44-45
Word-length registers, 36
WORD PTR operator, 56
Working set, 23, 173
Workstations, 11-17

CAD/CAM/CAE, 14-17
 scientific, 11
 speed comparisons (80286/80386), 30

Wraparound, 142
Writable (W) bit, 176
Write-back updating, 266-267
Write/Read (W/R#) signal, 236
Write-through updating, 266-267

X

XCHG instruction, 61, 102
XLAT instruction, 61, 103
 segment override, 159
XOR instruction, 59, 92, 115
 clearing registers, 59

Z

Zero-extended conversions, 61
Zero flag (ZF), 38
 bit scan instructions, 80
 carry from INC, 90
 string instructions, 107
 use, 38
Zero in front of hex numbers, 34
Zero iterations case, 101
Zero (null) selector, 165, 166

Trademarks

Unix is a registered trademark of AT&T.

Sidekick is a registered trademark of Borland International, Inc.

System V/386 is a trademark of Intel Corporation.

VP/ix is a trademark of INTERACTIVE Systems Corporation and of Phoenix Technologies, Ltd.

Locus is a registered trademark and Merge 386 is a trademark of Locus Computing Corporation.

Lotus and 1-2-3 are registered trademarks of Lotus Development Corporation.

Microsoft, MS-DOS, and XENIX V/386 are registered trademarks of Microsoft Corporation.

Prokey and RoseSoft are trademarks of RoseSoft, Inc.

About the Author

Lance A. Leventhal is an independent consultant specializing in microprocessors and personal computers. He has his own firm, Emulative Systems Company, in San Diego, CA. He has helped develop many microprocessor-based systems, including communications controllers, navigation systems, signal processors, and instruments. He has served as a consultant for Rockwell International, Anderson-Jacobson, NCR, NASA, Disney, and Universities Space Research Association.

Dr. Leventhal's previous experience includes affiliations with Linkabit Corporation, Intelcom Rad Tech, Naval Electronics Laboratory Center, and Harry Diamond Laboratories. He received a B.A. degree from Washington University (St. Louis, MO) and M.S. and Ph.D. degrees from the University of California, San Diego. He is a member of SCS, ACM, IEEE, IEEE Computer Society, and ASEE.

Also in The PC Library

The PC Configuration Handbook: A Complete Guide to Assembling, Enhancing, and Maintaining Your PC, by John Woram. Covers IBM PCs, ATs and Compatibles.

Master the Powerful World of the 80386!

Lance Leventhal's 80386 Programming Guide is a general reference manual for programmers, technicians, systems analysts, teachers, students, and hackers. Noted author, Lance Leventhal guides you through the 80386 step-by-step from its advanced 32-bit architecture to its most powerful multi-tasking functions.

Regardless of whether you use an 80386-based system in technical, industrial, or business applications, this invaluable sourcebook provides an in-depth, easy-to-understand discussion of all major aspects of 80386 software development.

The discussion includes:

- A task-oriented overview of the 80386's instruction set and assembly language.
- An explanation of I/O methods and descriptions of the common I/O chips.
- Thorough inspection of the 80386's memory and task management facilities.
- A functional description of the 80386 hardware.

For the reader with some exposure to the 8088 family and assembly language programming, this book explains the 80386's key features and the differences between it and earlier chips. It provides a basic understanding of how the 80386 works and what its capabilities will mean to you in real applications.

Lance Leventhal is the author of over 20 computer books with more than 700,000 copies in print. He is best known for his very popular series of chip books—on the 8080, 6800, 6809, and 68000 microprocessors—of which this book is a logical successor.

"The 80386 makes personal computers come of age, and this book helps you to master all its capabilities. The 80386 gives PCs the computing power and memory capacity of large machines. It will lead the way to bringing large database applications, financial models, CAD/CAM, artificial intelligence, robotics, and signal and image processing to the desktop."

—Lance A. Leventhal



9780553345292

03/28/2018 7:58-2

22

9780553345292-X>2195

34529-X ■ IN U.S. \$21.95 (IN CANADA \$26.95) ■ BANTAM COMPUTER BOOKS